A High-Level Comparison Between AP2 and Dyalog APL
Version 0.16
1/11/2026

Most of the material on the Internet comparing Dyalog APL with IBM's APL2 is limited in detail for two reasons.

First the online material is dated as Dyalog APL kept on moving forward closing more gaps and offering more features.

Second is that much of existing material is about migrating, which means it primarily focuses on the minimum needs to move forward from one version to the other and not all the extra stuff that has been added to Dyalog APL.

Some material at Dyalog.com provides some ideas but is not complete.

Note that some third-party reference material may assign names to functions or operations which are not the names in use, nor explained as such, in the documentation for the associated dialect.

This summary is based on the Unicode version of Dyalog APL. Additional differences may exist with the Classic version. In addition, Dyalog APL changes some of its behavior to be more like that of APL2 using the system of the Migration Level (⎕ML) system variable and the material here assumes the default value is used.

It is also based on a general focus on IBM's APL2 as a language with limited information about specific OS versions (e.g., PC, 370, etc.). It does not address Log-On APL2, the successor to IBM APL2, due to the lack of publicly available documentation.

This document is not intended to replace the material in other manuals. In some cases it only provides limited information about an item. In other cases, it provides more details to get a concept across.

Nomenclature of names and symbols

Even within an implementation, the names for a symbol may vary based on the context of the discussion (e.g., name of the symbol, name of the symbol when used as a monadic function, name of the symbol when used as a dyadic function, operator, etc.). The distinctions are not captured here, but many of the different names for each of them are.

Symbol	Dyalog APL	APL2
'	quote	quote apostrophe
⍘	minus negative subtract negate	negative subtract bar
!	exclamation mark factorial binomial	exclamation mark factorial binomial quote dot shriek
#	hash root root object	pound
##	parent parent object	
&	ampersand spawn	ampersand
(	left parentheses left round parenthesis	left paren opening parenthesis
(I)←	assignment(selective)	
)	right parentheses right round parenthesis	right paren closing parenthesis
*	star exponential power	star exponential power
,	comma ravel catenate laminate	comma ravel catenate laminate join
.	dot inner product name separator decimal point	dot inner product outer product array product operator
/	slash reduce replicate compress	slash reduce replicate compress
:	colon label colon end of label start control word	colon

Symbol	Dyalog APL	APL2
;	semicolon name separator array separator	semicolon
?	roll deal question mark	roll deal question mark query
@	at	commercial at
[	left bracket left square bracket	left bracket
[]	left bracket indexing axis	bracket index
[]←	assignment indexed modified	
[I]←	assignment(indexed)	
\	expand slope scan	expand slope scan backslash
]	right bracket indexing axis	right bracket bracket index
_	underbar character	underbar
{	left brace left curly brace	left curly brace
|	stile magnitude residue modulus absolute value remainder	stile magnitude residue modulus
}	right brace right curly brace	right curly brace
~	tilde not without excluding	tilde not without
¨	diaeresis each	dieresis each
¯	overbar negative high minus	overbar
+	add conjugate plus	add conjugate plus
⊖	reverse first rotate first	circle bar reverse rotate

Symbol	Dyalog APL	APL2
∘	jot beside bind	jot outer product
∘.	outer product	outer product
<	less less than	less less than
=	equal	equal
≠	not equal to unique mask not equal	not equal exclusive-or
>	greater greater than	greater greater than
×	multiply times direction signum	multiply times direction signum
÷	divide reciprocal	divide reciprocal
∆	delta	delta
∇	function self del editor del	del
∇∇	operator self	
∈	epsilon enlist membership type	epsilon enlist member
^	and logical and lowest common multiple caret pointer	and up caret
^	and logical and lowest common multiple caret pointer	
∨	or logical or greatest common divisor	or down caret
∩	up shoe intersection	down shoe
∪	set union unique union	up shoe
⍤		dieresis dot dotted del
≡	depth match equal underbar	depth match equal underbar

Symbol	Dyalog APL	APL2
≠	equal underbar slash tally not match	
≤	less or equal less than or equal to	not greater than less than or equal material implication
≥	greater or equal greater than or equal to	not less than greater than or equal
⊂	enclose partition left shoe partitioned enclose	enclose partition left shoe
⊃	disclose right shoe pick first	disclose right shoe pick
⊆	left shoe underbar nest partition	
⊢	right tack identity same right	left tack
⊣	left tack same left	right tack
⊤	encode representation down tack	encode up tack top
⊥	decode base up tack	decode down tack base value
⋄	diamond statement separator	diamond
⌈	upstile ceiling maximum round up	up stile ceiling maximum
⌊	downstile floor minimum round down	down stile floor minimum
⍞	quad evaluated input evaluated input/output	quad evaluated input/output box
⌶	i-beam ibeam	i-beam base top
⌷	squad index	squad index

Symbol	Dyalog APL	APL2
	materialise	
⌸	quad equal key	
⌹	matrix inverse matrix divide domino	matrix inverse matrix divide domino quad divide
⍁	quad diamond stencil	
⍜		quad jot
⍘	reverse rotate	reverse rotate circle stile
⍁	slash bar replicate first reduce first	slash bar slash reduce replicate
⍀	slope bar expand first scan first	slope bar backslash expand scan
⍂		quad slope quad backslash
⍉	transpose circle backslash	transpose circle slope
⍋	grade up	grade up delta stile grade up with collating sequence
⍎	execute	execute down tack jot base jot
⍒	grade down	grade down del stile grade down with collating sequence
⍕	thorn format	up tack jot format format by specification format by example top jot
⍙	delta underbar	delta underbar
⍝	comment lamp	comment lamp up shoe jot
⍞	quote-quad character input/output	quote quad character input/output quad quote

Symbol	Dyalog APL	APL2
		quad prime
⍟	logarithm natural logarithm	logarithm natural logarithm circle star
⍠	variant	
⍫	<no name>	del tilde
⍣	star diaeresis power operator	
⍤	jot diaeresis atop rank	
⍥	circle dieresis over	
⍨	tilde diaeresis commute constant	
⍪	comma bar table catenate first laminate	
⍬	zilde	
⍱	nor logical nor	nor down caret tilde
⍲	nand logical nand	nand up caret tilde
⍳	iota index of index generator	iota index of interval
⍴	rho shape reshape	rho shape reshape
⍷	epsilon underbar find	epsilon underbar find
⍸	iota underbar where interval index	iota underbar
⍺	left argument	alpha
⍵	right argument	omega
⍵⍵	right operand	
⍺⍺	left operand	
⍜	circle pi times circular	circle pi times
↑	up arrow take mix	up arrow take first
→	branch abort	branch arrow right arrow

Symbol	Dyalog APL	APL2
		escape
↓	down arrow drop split	down arrow drop
←	left arrow assignment	left arrow assignment arrow specification arrow
⍥	behind	

Characters

There are different symbols in the default character sets available for APL2 and Dyalog APL. For example (this may not be a complete list), the following symbols are only available in Dyalog APL:

≠ ⊆ ⌸ ⌺ ⌹ ⍣ ⍤ ⍥ ⍨ ⍪ ⍬ ⍥ ∧

Note: Dyalog APL supports (and honors) both ^ (circumflex accent; U+005E) and ∧ (logical and; U+2227) while APL2 only supports the first of these. This does not occur in all cases, such as when anchoring a search string with ⎕S.

All the ASCII-based APL2 characters are available in Dyalog APL, although they may not be as easy to enter nor operate in the same manner (for those which provide some functionality in APL2). For example, the symbols ⍨ ⍁ ⍟ are available in APL2, and while not valid tokens in Dyalog, they are valid characters which can be copied into Dyalog APL or obtained using ⎕UCS 8757 9026 9019.

Both APL2 and Dyalog have the capability to work with characters other than those in ⎕AV. In APL2 these are called extended characters. Some APL2 environments can provide a character set identification that defines a range of extended characters that can be correctly handled. In Dyalog APL, these are called Unicode characters and are available.

While APL2 has a ⎕UCS function like that in Dyalog APL, unlike Dyalog APL, APL2 characters that are not in ⎕AV cannot be displayed on most devices and are shown as the omega (⍵) character.

Underbarred Letters

In the default configuration, neither APL2 nor Dyalog supports underbarred characters. These have been mapped in each to other characters in the APL alphabet. The actual mapping is different between the two implementations.

Dyalog APL provides a means to utilize the underbarred characters, while APL2 does not.

```
      ⎕AVU[97+⍳26]←9397+⍳26
      ⎕AV[97+⍳26]
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The first statement above specifies that the relevant part of the atomic vector should be mapped to the circled alphabet in Unicode, the second displays the "third alphabet". What is displayed with Dyalog APL is dependent on the font in use. With the default font, the underlined characters are displayed. In any other Unicode font, these would display as the circled alphabet, Ⓐ to Ⓩ.

ASCII-based implementations of APL2 (such as APL2/PC, APL2/2, APL2 for Sun Solaris, and APL2/6000) do not define underbarred letters at all. Certain settings in APL2/370 (e.g., CMS and TSO) default to using underbarred uppercase letters instead of lowercase letters using the **CASE(**n**)** parameter. This parameter identifies the cases of the alphabets to be used when APL2 returns or displays the names of APL objects. For more information see APL2/370 Programming: System Services Reference.

When using **CASE**, n is one of the following:

0 Both lowercase and underbarred characters are valid and synonymous when evaluating names. Primitives that return names as results (⎕NL, ⎕CR, ⎕FX, ⎕SVQ, ⎕TF), and system commands and messages that produce names, produce them using the underbarred rather than the lowercase alphabet.

1 Both lowercase and underbarred characters are valid and synonymous when evaluating names. Primitives that return names as results, and system commands and messages that produce names, produce them using the lowercase rather than the underbarred alphabet.

2 In general, underbarred characters are not valid in names and are not accepted or produced in system functions, commands, or messages. Underbarred letters in the arguments to)COPY,)PCOPY,)MCOPY and)IN are accepted as lowercase letters to aid migration.

The default setting as shipped by IBM is **CASE**(1). An installation can change the default or specify an overriding value.

Note: The response to)ERASE, given underbarred letters, is the underbarred letters.

The CASE(n) invocation parameter does not apply to all work done by you during the APL2 session. Instead, it is interpreted as an implicit parameter of a)CLEAR command. CASE is, in fact, a workspace attribute in all workspaces. (CASE(0) was the default for workspaces created by APL2 Version 1.) The workspace attribute is saved with the workspace and is not changed when the workspace is loaded.

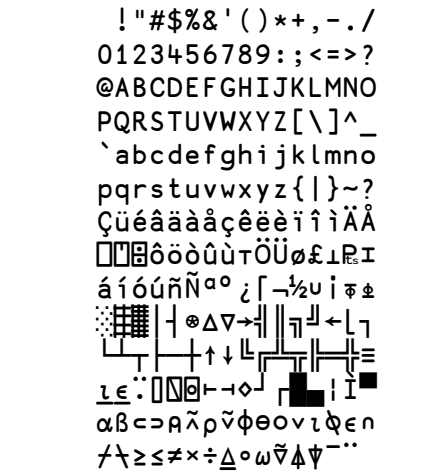
Objects may be copied into the active workspace using)COPY,)PCOPY,)MCOPY or)IN without affecting the workspace attribute. The names of copied objects, as well as names referred to by copied functions or defined operators, are converted appropriately.

Note: Literal strings and comments within functions, and the content of indirect copy lists, are not converted.

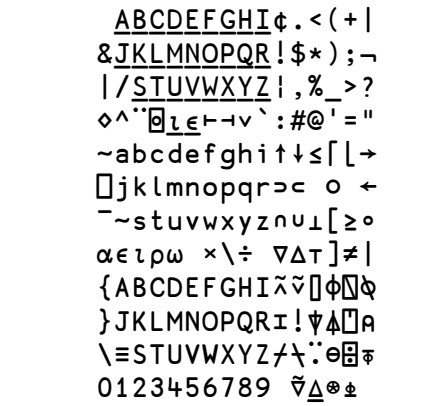
No matter what CASE setting or workspace attribute is in effect, shared variable names containing underbarred characters are never transferred from the APL2 interpreter to the shared-variable processor. The underbarred characters are instead translated to lowercase.

⎕AV

APL2 uses two primary values of ⎕AV, an ASCII-based value on personal computers, and an EBCDIC-based value on mainframes. The displayable ASCII-based value is:

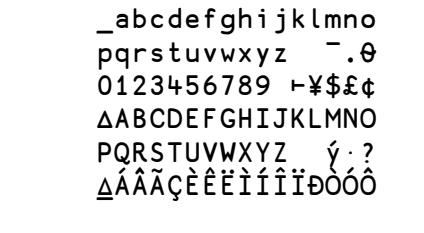


The EBCDIC-based value is:



A number of characters in APL2 will vary based on the national language in use at the time.

⎕AV is obsolete in Dyalog APL since the Unicode Edition was introduced and is only retained for backwards compatibility.[2] It no longer contains all the glyphs used for primitive functions and operators. The default value is:



ÕÙÚÛÝþãìðòõ{€}¬□
¨ÀÄÅÆ˜ÉÑÖØÜßàáâä
åæçèéêëíîïñ[/⍀\⍀
<≤=≥>≠∨∧−+÷×?∈⍴~
↑↓⍳○*⌈⌊∇⍬(⊂⊃∩∪⊥⊤
|;,⍫⍲⍱⍲⍱⍟⌸!⍒⍋⍫
¯≡≠óôöø"# &´⌋┐┌└
┼├┤┴┬|@ùúû^ü`¦¶
:≤¿¡⋄←→⍵)] §□□⍕

Neither the contents of ⎕AV, nor the position of like characters are the same between APL2 and Dyalog APL.

In addition, especially with the EBCDIC version of APL2, many characters look alike but are not (e.g., single quotes, vertical bars and exclamation points).

Dyalog APL provides the ability to change the symbols of ⎕AV using ⎕AVU, but what is displayed is partially dependent on the font in use. Setting a position of ⎕AVU to a specific Unicode code point results in the corresponding position in ⎕AV to return and display the associated Unicode character. This can be used to change a subset of ⎕AV (e.g., the underbarred letters as shown above) or all of ⎕AV (e.g., to the values used in APL2-ASCII or APL2-EBCDIC implementations).

Other Nomenclature Differences

Dyalog APL uses the term strand, APL2 uses the term vector.

APL2 uses the term underbarred when referencing underbarred letters while Dyalog APL uses underscored when referencing letters and underbar for all other characters.

Dyalog APL uses the terms glyphs and symbols while APL2 uses the term symbols and characters, calling those not in ⎕AV extended characters.

Binding

The Dyalog APL binding rules are:

> For two entities X and Y that are adjacent in an expression (that is, XY), the binding strength between them and the result of the bind is shown in this table:

		Y						
		A	**F**	**H**	**MOP**	**DOP**	**DOT**	**IDX**
X	**A**	6 A	3 AF	3 AF	4 F		7 REF	4 A
	F	2 A	1 F	4 F	4 F			4 F
	H		1 F	4 F	4 F			4 H
	AF	2 A	1 F					
	MOP			4 ERR				
	DOP	5 MOP	5 MOP	5 MOP				
	JOT	5 MOP	5 MOP	5 MOP	4 F			
	DOT	6 ERR	5 MOP	5 MOP		6 ERR		
	REF	7 A	7 F	7 H	7 MOP	7 DOP		

	IDX	3 ERR	3 ERR	3 ERR				

where:

A : *Array, for example, `0 1 2 'hello' α ω`

F : *Function (primitive/defined/derived/system), for example, `+ - +.× myfn ⎕CR {α ω}`

H : *Hybrid function/operator, that is, `/ ⌿ \ ⍀`

AF : Bound left argument, for example, `2+`

MOP : *Monadic operator, for example, `¨ ⍨ &`

DOP : Dyadic operator, for example, `⍤ ⌸ ⍣ ⌺`

JOT : Jot, that is, compose/null operand `∘`

DOT : Dot, that is, reference/product `.`

IDX : square-bracketed expression, for example, `[α+⍳ω]`

ERR : Error

* indicates a "first-class" entity, which can be parenthesised or named

In this table:

- the higher the number, the stronger the binding
- an empty field indicates no binding for this combination; an error.

For example, in the expression a b.c[d], where a, b, c and d are arrays, the binding proceeds:

```
      a b . c [d]
       6 7 6 4      ⍝ binding strengths between entities

→     a (b.) c [d]
       0    7 4

→     a (b.c) [d]
       6     4

→     (a(b.c))[d]
```

In APL2, the hierarchy of binding strengths in descending order is:

Binding Strength	**What Is Bound**
Brackets	Brackets to what is on their left
Specification left	Left arrow to what is on its left
Right operand	Dyadic operator to its right operand (a function or array)
Vector	Array to an array
Left operand	Operator to its left operand
Left argument	Function to its left argument
Right argument	Function to its right argument
Specification right	Left arrow to what is on its right

For binding, the branch arrow behaves as a monadic function. Brackets and monadic operators have no binding strength on the right. Parentheses change the default binding.

This results in several differences, for example:

Command	APL2 Results	Dyalog APL Results
`10 20 30[2]`	`RANK ERROR`	`20`
`(10 20 30)[2]`	`20`	`20`
`A←9 10 11` `1 2 A[2] 3 4 5 6[3]`	`RANK ERROR`	`4`
`1 2 A[2] 3 4 5 6`	`1 2 10 3 4 5 6`	`2 3 4 5 6`
`3 A←5`	`sets A to 5 and returns 3 5`	`SYNTAX ERROR`

`A←2 3 4`
`B←8`
`A B←'NEW'`
	`sets B to 'NEW" and returns 2 3 4 NEW`	`LENGTH ERROR`

`A←'HAT'`
`B←4 8 10`
`C←'W' 2 'X' 7`
`A[1] B[2] C[3]`
	`H 8 X`	`INDEX ERROR`

`A B C[2]`	`HAT 4 8 10 2`	`4 8 10`

Some implications:

- In APL2 bracket indexing binds stronger than stranding (vector), while the opposite is true in Dyalog APL. For example, the expression `a b c[d]` binds as `(a b c)[d]` in Dyalog APL (and other pre-nested array APLs such as APL\360) rather than as `a b (c[d])`, which is preferred by APL2.

- In Dyalog APL, strand notation has higher precedence than operator evaluation, while in APL2 this is only the case for the left operand, and the right operand is bound before stranding.

- In APL2, array-array binding is weaker than indexed-bracket and operator-array, but not array-operator. In Dyalog APL, array-array is almost the strongest.

- In APL2, operator-array binds stronger than array–array, which means that a literal vector right-operand must be parenthesized. This is opposed to Dyalog APL where stranding is stronger than right operand binding, which means that adjacent operands and arguments must be separated by parenthesis or function application.

- In Dyalog APL binding, if you accidentally fail to include the operator in parenthesis, strand binding will potentially treat it as a string. For example:

```
      ⍤
SYNTAX ERROR
      1 +⍤3 5
SYNTAX ERROR: Missing right argument
      1+⍤3 5
     +⍤3 5
┌─┬─┬───┐
│+│⍤│3 5│
└─┴─┴───┘
      1 (+⍤3) 5
8
```

System Commands

Dyalog APL	APL2	Command	Description	Notes
	X	)CHECK	Diagnose errors	
X		)CLASSES	List classes	
X	X	)CLEAR	Clear the workspace	What gets cleared and not cleared may vary between the two implementations. In APL2: - All shares are retracted. - Most system variables are set to their initial default values. Exceptions include ⎕NLT, ⎕PW and ⎕TZ. - Settings established by)PBS and)EDITOR are not reset. In Dyalog APL: - System variables are initialized with their default values. - In GUI implementations of Dyalog, all GUI objects are expunged, discards any unprocessed events in the event queue are discarded and the properties of the Root object '.' are reset to their default values. - .NET objects in the session namespace ⎕SE are disconnected from .NET because)CLEAR closes the current .NET AppDomain.
X		)CMD)SH	Execute a Windows or Unix Command	The command supports redirecting expected input requests to a prompt window. APL parameters can be set allowing for the use of an alternate host command program for executing the command.
	X	)HOST	Execute a host system command	Entering)HOST with no parameters results in the display of the name of the host system.
X	X	)CONTINUE	Save a Continue workspace and terminate APL	In APL2: - The CONTINUE workspace is loaded automatically when APL2 is restarted unless the INPUT parameter is used in the invocation of APL2. - When a CONTINUE workspace is loaded, the latent expression (if any) is executed.

Dyalog APL	APL2	Command	Description	Notes
				In Dyalog APL: - The CONTINUE workspace is NOT automatically loaded when APL is restarted. - When a CONTINUE workspace is loaded, the latent expression (if any) is NOT executed.
X	X	)COPY	Copy objects from another workspace	In APL2, the only system objects that can be copied are ⎕CT, ⎕FC, ⎕IO, ⎕LX, ⎕PP, ⎕PR and ⎕RL. In Dyalog APL,)COPY can also be done against a session file (.dse) or a namespace.
X		)CS	Change current namespace	
X	X	)DROP	Delete a named workspace	
X		)ED	Edit object using Dyalog editor	
	X	)EDITOR	Query or select editor	
X	X	)ERASE	Erase object(s)	In APL2, if the list of named items to erase includes the name of a simple character scalar, vector, or matrix enclosed within parentheses, its rows are interpreted as APL2 names, and these objects are erased instead of the array itself. The array may also contain its own name, which is then erased as well. This form of erasing is called indirect erase.
X		)EVENTS	List events of GUI namespace or object	
X	X	)FNS	List user defined Functions	In APL2, a string can be passed to the command of the form "[first] [-] [last]" to restrict the output to only those names within the specified range. In Dyalog APL: - The results of this command also includes the names of global defined functions in the current namespace. - If a string is passed to the command, characters which are not valid for a name are ignored. - If a string is passed to the command, only the first name character is used.
X		)HOLDS	Display Held tokens	

Dyalog APL	APL2	Command	Description	Notes
	X	)IN	Read transfer file into active workspace	
X	X	)LIB	List workspaces in a directory	In Dyalog APL, multiple directory may be listed based on the value of the configuration variable WSPath.
X	X	)LOAD	Load a workspace	On some APL2 platforms, an additional parameter can be supplied to control the size of the active workspace when loading. Without the size parameter, the maximum workspace size is used. In Dyalog APL, - Holding the Ctrl key down while entering a)LOAD command or selecting a workspace from the session file menu causes the incoming latent expression to be traced. - Holding the Shift key down while selecting a workspace from the session file menu will prevent execution of the latent expression.
X		)METHODS	List methods in GUI namespace or object	
	X	)MCOPY	Migrate VS APL objects into active workspace	
	X	)MORE	List additional diagnostics	In the workstation version of APL2 (e.g., Unix and Windows), returns the result NO MORE INFORMATION to all errors.
	X	)MSG	Send message to another user	
	X	)NMS	List names	
X		)NS	Create a global Namespace	
X		)OBJECTS	List global namespaces	
X		)OBS	List global namespaces (alternative form)	
X	X	)OFF	Terminate the APL session	
	X	)OPR	Send message to the system operator	
X	X	)OPS	List user-defined Operators	In APL2, a string can be passed to the command of the form "[first] [-] [last]" to restrict the output to only those names within the specified range. In Dyalog APL: - The results of this command also includes the names of global

Dyalog APL	APL2	Command	Description	Notes
				defined functions in the current namespace. - If a string is passed to the command, characters which are not valid for a name are ignored. - If a string is passed to the command, only the first name character is used.
	X	)OUT	Write objects to transfer file	
	X	)PBS	Printable backspace character	
X	X	)PCOPY	Perform Protected Copy of objects	
	X	)PIN	Read transfer file into active workspace	
X		)PROPS	List properties of GUI namespace or object	
	X	)QUOTA	Display resource limits	
X	X	)RESET	Reset the state indicator	In Dyalog APL,)RESET also performs an internal re-organization of the workspace and process memory: 1. Any un-referenced memory is discarded. 2. Numeric arrays are demoted to their tightest internal form. 3. All remaining used memory blocks are copied to the low-address end of the workspace, leaving a single free block at the high-address end. 4. All memory allocated is returned to the Operating System except the space required for the compacted workspace, plus a working overhead based on the configured maximum workspace size (MAXWS).
X	X	)SAVE	Save the active workspace	
X		)SH	Execute UNIX command	
X	X	)SI	State Indicator	The information displayed about suspended functions and operators varies between the APL2 and Dyalog APL.
X		)SIC	Clear State Indicator	
X	X	)SINL	State Indicator with local Name Lists	The information displayed about suspended functions and operators varies between the APL2 and Dyalog APL.
	X	)SIS	Display state indicator	
	X	)SYMBOLS	Query/modify symbol table size	
X		)TID	Switch current Thread Identity	

Dyalog APL	APL2	Command	Description	Notes
	X	)TIME	Display current time	
X	X	)VARS	List user defined global Variables	In APL2, a string can be passed to the command of the form "[first] [-] [last]" to restrict the output to only those names within the specified range. In Dyalog APL: - The results of this command also includes the names of globally defined variables in the current namespace. - If a string is passed to the command, characters which are not valid for a name are ignored. - If a string is passed to the command, only the first name character is used.
X	X	)WSID	Query/assign workspace identifier	
X		)XLOAD	Load a workspace; do not execute ⎕LX	

User Commands

Dyalog APL supports a class of commands whose availability is similar to those of system commands. These differ from system commands in that the underlying implementation code is text-based APL and can be created by an user. They begin with the right-bracket symbol. Several user commands are supplied with Dyalog.

User Command	Category	Description
]Activate	Dev & present tools	Activate or reset an experimental tool shipped with Dyalog APL
]Add	Link	Associate items in linked namespaces with new files/directories in corresponding directory, optionally with simultaneous definition
]ADoc	Dev & present tools	Automated documentation generation
]Align	Fns & Ops	Align end-of-line comments
]APLCart	Dev & present tools	Access a searchable collection of over 3000 short APL phrases
]Assemblies	MS Window	List all .NET assemblies currently loaded into memory
]Boot	SALT	Boot from a file containing instructions or a function
]Box	Output	Display output with borders indicating shape, type and structure
]Boxing	Output	Display output with borders indicating shape, type and structure
]Break	Link	Break link between namespace and corresponding directory
]Calendar	Dev & present tools	Display calendar
]Calls	Fns & Ops	Produce the calling tree of a function in a class/namespace/scriptfile
]Caption	MS Window	Query or change window captions

User Command	Category	Description
]CD	File	Report (and, optionally, change) the current directory
]Chart	Dev & present tools	Run the chart wizard on argument APL expression
]Check	Workspaces	Perform workspace integrity check
]Clean	SALT	Remove SALT tags from the workspace or specific items so that SALT no longer saves changes to associated files
]Collect	File	Merge all the files with the specified path/name{-nnn} into a single file
]Compare	Array	Compare two variables/arrays
]Compare	File	Compare two component files
]Compare	Fns & Ops	Compare two functions
]Compare	SALT	Compare two versions of a SALTed item
]Compare	Workspaces	Compare two workspaces
]Config	Dev & present tools	Display value of configuration parameters (optionally with origin)
]Configure	Link	Set directory or user configuration options
]CopyReg	MS Window	Copy registry entries to file and, optionally, between Dyalog instances
]Create	Link	Link a namespace with a directory (create one but not both if non-existent)
]DBuild	Devops	Run one or more DyalogBuild script files (.dyalogbuild)
]Deactivate	Dev & present tools	Deactivate a previously activated tool
]Defs	Fns & Ops	List single-line dfns, dops, derived functions and trains
]Demo	Dev & present tools	Provide playback mechanism for live demonstrations
]DInput	Fns & Ops	Define a multi-line dfn, dop or derived function/operator, or execute a multi-line expression
]Disp	Output	Display specified array with borders indicating sub-array shape and type
]Display	Output	Display specified array with borders indicating array and sub-array shape and type
]Document	Workspaces	List (part of) the workspace content, displaying each item separately
]DTest	Devops	Run (a selection of) functions named test_* from a namespace, file or directory
]Edit	Array	Edit a simple or mixed array of enclosures using the Array Editor from davidliebtag.com
]Edit	File	Open the specified native file as an editable text file
]Export	Link	Export a namespace to a directory (create the directory if absent); does not create a link
]Expunge	Link	Erase item and associated file
]Factors	Calc	Determine the prime factors of the argument
]FileAssociations	MS Window	Configure Microsoft Windows to associate directories and files with a specific Dyalog instance (optionally through a GUI)
]Find	File	List line numbers and lines that contain the specified string for each file that has a match
]Find	Output	Precede output with a reference to the line of code that generated it

User Command	Category	Description
]FindRefs	Workspaces	Follow references in the workspace until all references have been found
]FnsLike	Workspaces	List functions and operators matching a pattern
]Format	Output	Format text into vector of text vectors (or matrix if called from Session) to fit ⎕PW using specified margins
]FromHex	Calc	Convert a list of hexadecimal representations of integers to a numeric vector
]Get	Experimental	Fetch data/code in many formats from local or remote sources
]GetFileName	Link	Return name of file associated with item
]GetItemName	Link	Return name of item associated with file
]GetTools4CITA	Devops	Load tools to run CITA tests
]GUIProps	MS Window	List property names and values for given or current object
]Help	Dev & present tools	Browse or get link to information about any APL concept
]HTML	Output	Render HTML/SVG or a website
]Import	Link	Import a namespace from a directory (create the namespace if absent); does not create a link
]In	WS Transfer	Import a workspace from a workspace transfer file
]KeyPress	MS Window	Return message arguments of KeyPress events
]LastResult	Output	Record Last session output
]Latest	Fns & Ops	List functions modified since a specified date (default is today), most-recently-changed first
]Layout	Output	Format text into vector of text vectors (or matrix if called from Session) to fit ⎕PW using margins inferred from the text
]LINK	Link	User commands for namespace-directory synchronization (see https://dyalog.github.io/link):
]List	SALT	List files (default: .dyalog only) and directories in the specified directory
]Load	SALT	Load item from native text file
]Locate	Workspaces	Locate (and, optionally, replace) strings in the workspace
]Map	Workspaces	Display namespace treeview
]Names	Workspaces	List names
]NamesLike	Workspaces	List names followed by their class matching the pattern
]Nms	Workspaces	List names followed by their class matching the pattern
]ObsLike	Workspaces	List objects matching the pattern
]Open	File	Open a file or directory with the specified application
]Out	WS Transfer	Export the current workspace to a workspace transfer file
]Peek	Workspaces	Execute expression in temporary copy of workspace
]PivotTable	Calc	Create a pivot table from an appropriate matrix
]Plot	Output	Plot data
]Profile	Performance	Report performance details (optionally through a GUI)
]Refresh	Link	Manually synchronise namespace or directory contents
]Refresh	SALT	Reload all SALTed items from their associated files
]RemoveVersions	SALT	Remove one or more versions of a file managed by SALT
]ReorderLocals	Fns & Ops	Sort local names in the header of Tradfns and Tradops
]Replace	File	Replace strings in files and return the number of changes made

User Command	Category	Description
]Repr	Output	Represent given value as APL/APLAN/JS/JSON/XML/CSV/SSV/PSV/TSV
]Resync	Link	Automatically synchronise namespace-directory differences
]Rows	Output	Cut, wrap, fold or extend the display of output lines to fit the Session window
]RunTime	Performance	Report execution time of one or more expressions
]Save	SALT	Save item in a native text file (default: same place if already SALTed)
]ScriptUpdate	Namespace	Synchronise namespace/class script to match current content
]Set	SALT	Return one or all parameters or set one parameter
]Settings	SALT	Return one or all parameters or set one parameter
]SizeOf	Workspaces	Report size of the current namespace's variables in descending order
]Snap	SALT	Save all new or modified items to native text files (unscripted namespaces become directories)
]SpaceNeeded	Performance	Compute memory needed to run expression(s)
]Split	File	Split a single file into (up to 999) smaller files
]Status	Link	List active namespace-directory links
]Stop	Link	Set, clear or report on breakpoints
]Summary	Namespace	Summarize (scope, size, syntax) the functions in a namespace/class/scriptfile
]ToHex	Calc	Convert integer(s) to a vector of text vectors containing the hexadecimal representation of each number
]ToLarge	File	Transform component files in the specified directory from small-span to large-span
]ToQuadTS	File	Convert a component file timestamp (single float number) to ⎕TS format (vector of 7 numbers)
]Touch	File	Check whether the specified file exists in the current/specified location and create it if it cannot be found
]Trace	Link	Set, clear or report on lines traced
]UDebug	User command	Facilitate debugging of user commands
]ULoad	User command	Load a user command's script into the current namespace
]UMonitor	User command	Gather user command execution data
]UNew	User command	Create one or more new user commands (optionally using a GUI)
]UReset	User command	Refresh cache of all user command definitions
]USetup	User command	Run Setup from setup.dyalog
]UVersion	User command	Report version information of a user command
]VarsLike	Workspaces	List variables matching the pattern
]Version	Dev & present tools	Report version numbers of APL, OS, SALT, UCMD, .NET and/or workspace
]View	Output	Evaluate and view in a read-only Editor window
]Xref	Namespace	Describe the inter-object cross-references in a class/namespace/scriptfile

System Variables and Functions

The table below provides a comparison of the system functions and variables in each implementation.

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL	APL2	⎕	Evaluated Input/Output	
Dyalog APL	APL2	⍞	Character Input/Output	In APL2, If the ⍞ request was preceded by one or more assignments to ⍞ without any intervening input or output, the last line of all the output characters, known as the prompt, is returned as the prefix of the response according to a number of rules: - If ⎕PR is null, the prompt (as may have been changed by the user) is used as the prefix. - If ⎕PR is not null, then any characters not changed in the prompt will be replaced by the value of ⎕PR in the prefix, changed characters will be returned as entered and any characters in the prompt past those which the user may have added or deleted will be as they appeared in the prompt. In Dyalog APL, If the ⍞ request was preceded by one or more assignments to ⍞ without any intervening input or output, the last line of the output characters (as may have been changed by the user) is returned as the prefix of the response. In both dialects, the last line is the last displayed line as controlled by ⎕PW. With the default settings, this means that APL2 returns a prefix of spaces equal in length to that of the prompt while Dyalog APL returns the original prompt. With APL2, setting ⎕PR to null makes it behave more like Dyalog APL.
Dyalog APL		⎕A	Alphabetic upper-case characters	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL		⎕Á	Underscored Alphabetic Characters	Returns characters with the same index in ⎕AV as was previously used for the underscored alphabetic characters. The variable is now ⎕Á but was previously ⎕A. What is displayed is dependent on the particular APL font is use. With later fonts, it is: ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝþãìðòõ
	APL2	⎕AF	Atomic Function	
Dyalog APL	APL2	⎕AI	Account Information	In APL2 workstation implementations, the contents of ⎕AI are incorrect after they reach 2*31. This can occur for sessions active longer than 24 days.
Dyalog APL		⎕AN	Account Name	
Dyalog APL		⎕ARBIN	Arbitrary Input	
Dyalog APL		⎕ARBOUT	Arbitrary Output	
Dyalog APL	APL2	⎕AT	Object Attributes	In APL2, the left argument is required. In Dyalog APL, when used as a monadic function, results are returned about the function headers of the arguments. The results for the left argument of 4 are different between the two implementations. APL2 returns 0 for functions and objects while dialog does not. In addition, how size is calculated also differs.
Dyalog APL		⎕ATX	Extended Attributes	
Dyalog APL	APL2	⎕AV	Atomic Vector	Order and contents are not the same between the implementations. In Dyalog APL, the contents can be modified thru the use of ⎕AVU.
Dyalog APL		⎕AVU	Atomic Vector - Unicode	
Dyalog APL		⎕BASE	Base Class	
Dyalog APL		⎕C	Case Convert	
Dyalog APL		⎕CLASS	Class	
Dyalog APL		⎕CLEAR	Clear Workspace (WS)	
Dyalog APL		⎕CMD	Execute the Windows Command Processor or another program	
Dyalog APL		⎕CMD	Start a Windows Auxiliary Processor	
Dyalog APL	APL2	⎕CR	Canonical Representation	
Dyalog APL		⎕CS	Change Space	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL		⎕CSV	Comma Separated Values	
Dyalog APL	APL2	⎕CT	Comparison Tolerance	
Dyalog APL		⎕CY	Copy objects into active WS	
Dyalog APL		⎕D	Digits	
Dyalog APL		⎕DCT	Decimal Comp Tolerance	
Dyalog APL		⎕DF	Display Format	
Dyalog APL		⎕DIV	Division Method	
Dyalog APL	APL2	⎕DL	Delay	
Dyalog APL		⎕DM	Diagnostic Message	
Dyalog APL		⎕DMX	Extended Diagnostic Message	
Dyalog APL		⎕DQ	Await and process events	
Dyalog APL		⎕DR	Data Representation	
Dyalog APL		⎕DT	Datetime	
	APL2	⎕EA	Execute Alternate	
	APL2	⎕EC	Execute Controlled	
Dyalog APL		⎕ED	Edit one or more objects	
Dyalog APL	APL2	⎕EM	Event Message (APL2) Event Messages (Dyalog)	⎕EM represents completely different items in the two implementations. In APL2, it represents a 3-row matrix of the error message, code line and pointer associated with the first line of the state indicator. In Dyalog APL, it returns the message(s) associated with the integer right argument.
Dyalog APL		⎕EN	Event Number	
	APL2	⎕ES	Event Simulation	
	APL2	⎕ET	Event Type	
Dyalog APL	APL2	⎕EX	Expunge objects	APL2 supports expunging the following: ⎕CT, ⎕FC, ⎕IO, ⎕LX, ⎕PP, ⎕PR and ⎕RL. Dyalog APL supports passing a vector of names.
Dyalog APL		⎕EXCEPTION	Reports the most recent Microsoft .NET Exception	
Dyalog APL		⎕EXPORT	Export objects	
Dyalog APL		⎕FAPPEND	Append a component to File	
Dyalog APL		⎕FAVAIL	File system Availability	
	APL2	⎕FC	Format Control	
Dyalog APL		⎕FCHK	File Check and Repair	
Dyalog APL		⎕FCOPY	Copy a File	
Dyalog APL		⎕FCREATE	Create a File	
Dyalog APL		⎕FDROP	Drop a block of components	
Dyalog APL		⎕FERASE	Erase a File	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL		⎕FHIST	File History	
Dyalog APL		⎕FHOLD	File Hold	
Dyalog APL		⎕FIX	Fix	
Dyalog APL		⎕FLIB	List File Library	
Dyalog APL		⎕FMT	Format	
Dyalog APL		⎕FNAMES	Names of tied Files	
Dyalog APL		⎕FNUMS	Tie Numbers of tied Files	
Dyalog APL		⎕FPROPS	File Properties	
Dyalog APL		⎕FR	Floating-Point Representation	
Dyalog APL		⎕FRDAC	Read File Access matrix	
Dyalog APL		⎕FRDCI	Read Component Information	
Dyalog APL		⎕FREAD	Read a component from File	
Dyalog APL		⎕FRENAME	Rename a File	
Dyalog APL		⎕FREPLACE	Replace a component on File	
Dyalog APL		⎕FRESIZE	File Resize	
Dyalog APL		⎕FSIZE	File Size	
Dyalog APL		⎕FSTAC	Set File Access matrix	
Dyalog APL		⎕FSTIE	Share-Tie a File	
Dyalog APL		⎕FTIE	Tie a File exclusively	
Dyalog APL		⎕FUNTIE	Untie Files	
Dyalog APL	APL2	⎕FX	Fix definition	APL2 supports a dyadic version in which execution properties are set using the left argument (displayability, suspendability, attention signal ignorance, conversion of nonresource error to DOMAIN ERROR). Dyalog APL supports additional forms for its right argument such as nested representation form and object representation form.
Dyalog APL		⎕INSTANCES	Instances	
Dyalog APL	APL2	⎕IO	Index Origin	
Dyalog APL		⎕JSON	JSON Convert	
Dyalog APL		⎕KL	Key Labels	
	APL2	⎕L	Left Argument	In APL2 workstation implementations, ⎕L is not recognized and returns a VALUE ERROR.
Dyalog APL	APL2	⎕LC	Line Counter	
Dyalog APL		⎕LOAD	Load a saved WS	
Dyalog APL		⎕LOCK	Lock a function	
Dyalog APL	APL2	⎕LX	Latent Expression	
Dyalog APL		⎕MAP	Map a file	
Dyalog APL		⎕MKDIR	Create a directory	
Dyalog APL		⎕ML	Migration Level	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL		⎕MONITOR	Monitor query	
Dyalog APL		⎕MONITOR	Monitor set	
Dyalog APL	APL2	⎕NA	Name Association	⎕NA, name association, is significantly different between the implementations. As such their arguments and operations should be reviewed in associated documentation. In APL2, a name is association with a associated processor. There are four processors provided with APL2: - Processor 0, which is APL itself. - Processor 10 through REXX programs may be executed. - Processor 11 through which programs written in non-APL languages can be called. - Processor 12 provides facilities through which APL data files and operating system files can be accessed using normal APL2 syntax. In Dyalog APL, a name association provides access from APL to a compiled function within a library. A library can be one of: - a Dynamic Link Library (DLL) under Windows, - a Shared Library (.so or .dylib) under Linux or macOS, or - a static library (.a) under AIX.
Dyalog APL		⎕NAPPEND	Append to File	
Dyalog APL	APL2	⎕NC	Name Classification	In APL2, system variables and functions are identified as such (returning '2' and '3' respectively. In Dyalog APL, they are identified as invalid names (returning ¯1). In Dyalog APL, the argument can also be a vector of vectors. Dyalog APL also returns more categories then APL2. For simple arguments it can also return: - 8 Event - 9 Object (GUI, namespace, COM, .NET)

Dyalog APL	APL2	Name	Description	Notes
				If the argument is nested, a more precise name class is returned: 2.1 array-variable 2.2 array-field 2.3 array-property 2.6 array-external shared 3.1 function-traditional 3.2 function-dfn 3.3 function-derived or primitive 3.6 function-external 4.1 operator-traditional 4.2 operator-dop 4.3 operator-derived or primitive 9.1 namespace-created by KNS,)NS or :Namespace 9.2 namespace-instance 9.4 namespace-class 9.5 namespace-interface 9.6 namespace-external class 9.7 namespace-external interface
Dyalog APL		⎕NCOPY	Copy files and directories	
Dyalog APL		⎕NCREATE	Create a File	
Dyalog APL		⎕NDELETE	Delete a File or Directory	
Dyalog APL		⎕NERASE	Erase a File	
Dyalog APL		⎕NEW	New Instance	
Dyalog APL		⎕NEXISTS	Discover whether or not a file or directory exists	
Dyalog APL		⎕NGET	Read Text File	
Dyalog APL		⎕NINFO	Obtain information about one or more files and/or directories	
Dyalog APL	APL2	⎕NL	Name List	Dyalog APL includes an extended list of name classification values (see ⎕NL above) which can be passed as the right argument. If an integer value is passed, it includes all sub-categories identified by the associated non-integer ⎕NL values. In any value of the right argument is negative, all values are treated as if they were negative and the result is a vector of of character vectors instead of a simple character matrix. In Dyalog APL, ⎕NL may be used to obtain lists of the Methods, Properties and Events provided by certain Dyalog APL objects (e.g., GUI

Dyalog APL	APL2	Name	Description	Notes
				objects, .NET classes and COM objects).
Dyalog APL		⎕NLOCK	Lock a region of a file	
	APL2	⎕NLT	National Language Translation	In APL2 workstation implementations, ⎕NLT can be referenced, but not set. National language is set with the -nlt invocation option.
Dyalog APL		⎕NMOVE	Move files and directories	
Dyalog APL		⎕NNAMES	Names of tied Files	
Dyalog APL		⎕NNUMS	Tie Numbers of tied Files	
Dyalog APL		⎕NPARTS	Split a file name into its constituent parts.	
Dyalog APL		⎕NPUT	Write Text File	
Dyalog APL		⎕NQ	Place an event on the Queue	
Dyalog APL		⎕NR	Nested Representation	
Dyalog APL		⎕NREAD	Read from File	
Dyalog APL		⎕NRENAME	Rename a File	
Dyalog APL		⎕NREPLACE	Replace data on File	
Dyalog APL		⎕NRESIZE	File Resize	
Dyalog APL		⎕NS	Namespace	
Dyalog APL		⎕NSI	Namespace Indicator	
Dyalog APL		⎕NSIZE	File Size	
Dyalog APL		⎕NTIE	Tie a File exclusively	
Dyalog APL		⎕NULL	Null Item	
Dyalog APL		⎕NUNTIE	Untie Files	
Dyalog APL		⎕NXLATE	Specify Translation Table	
Dyalog APL		⎕OFF	End the session	
Dyalog APL		⎕OPT	Variant Operator	
Dyalog APL		⎕OR	Object Representation	
Dyalog APL		⎕PATH	Search Path	
Dyalog APL		⎕PFKEY	Programmable Function Keys	
Dyalog APL	APL2	⎕PP	Print Precision	In APL2, if ⎕PP is set to a value above the maximum for the current system, the new value is saved but the maximum is used (if the value is set to a very large value, it will be treated as if an invalid value was assigned). In Dyalog APL, if ⎕PP is set to 34 (when ⎕FR is 1287) or to a value ≥17 (when ⎕FR is 645), the maximum precision is used. Setting it to ≥34 results in a DOMAIN ERROR.
	APL2	⎕PR	Prompt Replacement	
Dyalog APL		⎕PROFILE	Profile Application	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL	APL2	⎕PW	Print Width Session	In Dyalog APL, ⎕PW only affects output, either direct or through ⎕ output. It does not affect the result of the function Format (⍕), of the system function ⎕FMT, or output through the system functions ⎕ARBOUT and ⎕ARBIN, or output through ⍞. If the Auto_PW parameter is set to 1, ⎕PW is automatically, and temporarily, adjusted whenever the Session window is resized. There may be additional rules regarding the resulting display that are unique to each implementation.
Dyalog APL	APL2	⎕R	Right Argument (APL2) Search and Replace (Dyalog APL)	⎕R represents two completely different things in APL2 and Dyalog APL. In APL2, when implemented, it represents the value of the right argument to the currently suspended operation in a suspended function. This allows the value to be retrieved, and potentially changed (which is sometimes not allowed, depending in the type of error), prior to completing the operation. In APL2 workstation implementations, ⎕R is not recognized and returns a VALUE ERROR. In Dyalog APL, it represents a search and replace operator.
Dyalog APL		⎕REFS	Local References	
Dyalog APL	APL2	⎕RL	Random Link	In APL2, ⎕RL is an integer used to set the random number seed. In Dyalog APL, ⎕RL is 2-element vector. Its first element contains the random number seed and its second element is an integer that identifies the random number generator that is currently in use. The allowable seed values vary based on the generator in use (e.g., 0, ⍬ or an integer of varying ranges).

Dyalog APL	APL2	Name	Description	Notes
				For backwards compatibility, Dyalog APL supports assigning a single value, which uses the value as a seed for the current generator. ⎕RL does not behave like a regular 2-element variable; it has its own rules relating to assignment and reference. ⎕RL may only be assigned in its entirety. Indexed and selective assignment used to assign values to individual elements may not work properly (and may not error). Depending on the algorithm in use, the first element itself may be a vector upon retrieval.
Dyalog APL		⎕RSI	Space Indicator	
Dyalog APL		⎕RTL	Response Time Limit	
Dyalog APL		⎕S	Search	
Dyalog APL		⎕SAVE	Save the active WS	
Dyalog APL		⎕SD	Screen Dimensions	
Dyalog APL		⎕SE	Session Namespace	
Dyalog APL		⎕SH	Execute a UNIX command or another program or start a UNIX AP	
Dyalog APL		⎕SHADOW	Shadow names	
Dyalog APL		⎕SHELL	Execute external program	
Dyalog APL		⎕SI	State Indicator	
Dyalog APL		⎕SIGNAL	Signal event	
Dyalog APL		⎕SIZE	Size of objects	
Dyalog APL		⎕SM	Screen Map	
Dyalog APL		⎕SR	Screen Read	
Dyalog APL		⎕SRC	Source	
Dyalog APL		⎕STACK	Report Stack	
Dyalog APL		⎕STATE	Return State of an object	
Dyalog APL		⎕STOP	Query Stop vector	
Dyalog APL		⎕STOP	Set Stop vector	
Dyalog APL	APL2	⎕SVC	Query & set Shared Variable (access) Control	
	APL2	⎕SVE	Shared Variable Event	
Dyalog APL	APL2	⎕SVO	Shared Variable Offer and Query degree of coupling	In APL2, shared variables are used between auxiliary processors, other users or APL2 interpreters, which are each identified by an assigned number unique to the processor or user.

Dyalog APL	APL2	Name	Description	Notes
				In APL2 workstation implementations, general shared variable offers are not supported. In Dyalog APL, shared variables are used between a workspace and another APL workspace or with another application, which is identified by a string of the form "protocol:application|topic". Both APL2 and Dyalog may have restrictions on shared variable names or surrogate values, based on the partner being communicated with. The two implementations differ in the coupling values of 0 and 1 for an offer. APL2 uses 0 for failed and 1 for pending while Dyalog APL uses 1 for failure and does not use 0.
Dyalog APL	APL2	⎕SVQ	Shared Variable Query	In Dyalog APL, ⎕SVQ is implemented for compatibility with other versions of APL but currently performs no useful function, returning a NONCE ERROR.
Dyalog APL	APL2	⎕SVR	Shared Variable Retraction	In APL2, the results represent the degree of coupling prior to the retraction for each corresponding value in the argument. In Dyalog APL, the results represent the degree of coupling after the retraction.
Dyalog APL	APL2	⎕SVS	Query Shared Variable State	
Dyalog APL		⎕TALLOC	Allocate Token Range	
Dyalog APL	APL2	⎕TC	Terminal Control Characters	In APL2, the value is the following three characters: backspace, new line (return), line feed. In Dyalog, the default value is the following three characters: backspace, line feed, new line (return). If ⎕ML≥3 the order of the elements is instead compatible with APL2 (in addition to other changes).
Dyalog APL		⎕TCNUMS	Thread Child Numbers	
	APL2	⎕TF	Transfer Form	
Dyalog APL		⎕TGET	Get Tokens	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL		⎕THIS	This Space	
Dyalog APL		⎕TID	Current Thread Identity	
Dyalog APL		⎕TKILL	Kill Threads	
Dyalog APL		⎕TNAME	Current Thread Name	
Dyalog APL		⎕TNUMS	Thread Numbers	
Dyalog APL		⎕TPOOL	Token Pool	
Dyalog APL		⎕TPUT	Put Tokens	
Dyalog APL		⎕TRACE	Query or Set Trace vector	
Dyalog APL		⎕TRAP	Event Trap	
Dyalog APL		⎕TREQ	Token Requests	
Dyalog APL	APL2	⎕TS	Timestamp	In APL2, the value of ⎕TS is offset from Greenwich Mean Time (GMT) according to the value of the Time Zone system variable (⎕TZ). In Dyalog APL, the value of ⎕TS is the clock time set on the particular system APL is installed on.
Dyalog APL		⎕TSYNC	Wait for Threads to Terminate	
	APL2	⎕TZ	Time Zone	
Dyalog APL	APL2	⎕UCS	Universal Character Set (APL2) Unicode Convert (Dyalog APL)	In APL2, most characters that are not in ⎕AV are shown as the omega (⍵) character. Dyalog APL supports a robust set pf Unicode characters set and can display a large set of these. In Dyalog APL, an optional left argument can be specified to convert between Unicode characters and one of three standard variable-length Unicode encoding schemes ('UTF-8', 'UTF-16', 'UTF-32').
	APL2	⎕UL	User Load	
Dyalog APL		⎕USING	Microsoft .NET Search Path	
Dyalog APL		⎕VFI	Verify and Fix numeric	
Dyalog APL		⎕VGET	Value Get	
Dyalog APL		⎕VR	Vector Representation	
Dyalog APL		⎕VSET	Value Set	
Dyalog APL	APL2	⎕WA	Workspace Available	In Dyalog APL, ⎕WA will also result in an internal reorganization of the workspace and process memory.
Dyalog APL		⎕WC	Create GUI object	
Dyalog APL		⎕WG	Get GUI object properties	
Dyalog APL		⎕WN	Query GUI object Names	
Dyalog APL		⎕WS	Set GUI object properties	

Dyalog APL	APL2	Name	Description	Notes
Dyalog APL		⎕WSID	Workspace Identification	
Dyalog APL		⎕WX	Expose GUI property names	
Dyalog APL		⎕WX	Window Expose	
Dyalog APL		⎕XML	XML Convert	
Dyalog APL		⎕XSI	Extended State Indicator	
Dyalog APL		⎕XT	Associate External variable	
Dyalog APL		⎕XT	Query External variable	

Some general observations:

- Assignment of values to a read-only system variable is ignored in APL2 and sometimes used as a sink function. In Dyalog APL this generates a `SYNTAX ERROR`.
- Dyalog APL's ⎕TRACE and ⎕STOP replace APL2 `T^FUNCTION` and `S^FUNCTION` capabilities. The windowed versions of each implementation also offer additional capabilities thru their respective menus.
- The rank of some system functions may differ between implementations. For example, in APL2, `⎕NC 'A'` returns a scalar, but Dyalog returns a one-element vector. Monadic ⎕SVO also exhibits this behavior.
- In Dyalog APL, several of the system functions include extended scope as it relates to namespaces.
- Default values for certain system variables are different (e.g., may be constrained by limitations in the underlying OS or platform).

Library and Workspace Names

In APL2:

> Workspace names may contain up to eight characters (A through Z, 0 through 9), and must begin with a character.
>
> In most commands where a workspace is specified, workspace names can be prefixed by library numbers which are associated with specific directories in the underlying operating system This association is not discussed here.
>
> The name of a workspace may specify a fully-qualified directory path name for a file.
>
> If the name of the file in which a workspace is saved contains spaces, the workspace name argument for associated system functions must be surrounded by two double-quote (") characters. To include a " character in the workspace name (on a non-Windows system), you must specify two adjoining double-quotes (that is, """").

In Dyalog APL:

> The name of a workspace must conform to file name conventions of the underlying operating system. This allows for an extended set of characters in a workspace name such as quotes and spaces.
>
> The name of a workspace may specify an operating system file suffix. When a workspace name is specified without a file suffix, these are added or implied. A system configuration parameter specifies the default suffix.
>
> The name of a workspace may specify a fully-qualified directory path name for a file.

If the name of the file in which a workspace is saved contains spaces, the workspace name argument for associated system functions must be surrounded by two double-quote (") characters. To include a " character in the workspace name (on a non-Windows system), you must specify two adjoining double-quotes (that is, """").

Under Windows, if no parameter is provided to the system commands associated with a workspace, the File Open dialog box is displayed.

In most situations where a workspace is specified, a workspace name can be prefixed by a directory path (using slashes or backslashes with no spaces, and this separator used between the path and the workspace name). This directory path starts with a sub-directory from the one which Dyalog APL was launched (under Windows this may not be the directory the program is in if it is launched using a shortcut).

When `)LOAD`, `)COPY` or `)PCOPY` is issued, and when you start an Auxiliary Processor without explicitly specifying a path in the name, Dyalog APL may search multiple directories for the named workspace if there are multiple directories identified in the configuration variable WSPath. `)LIB` without any parameters will be done for all the directories in WSPath.

Legal Names

In APL2, legal user-constructed object names (e.g., variables, defined functions, defined operators and labels) may be any sequence of characters, starting with a character from the first two rows , selected from the following:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZΔ
abcdefghijklmnopqrstuvwxyzΔ
0123456789_¯
```

Notes:

Certain configuration settings in APL2/370 (e.g., CMS and TSO) can be set to default to using underbarred uppercase letters instead of lowercase letters

`SΔ` (Stop Control) and `TΔ` (Trace Control) cannot be used as the first two characters of a name as they are reserved for the identified functionality.

In Dyalog APL, legal user-constructed object names may be any sequence of characters, starting with a non-numeric character, selected from the following:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_
abcdefghijklmnopqrstuvwxyz
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝß
àáâãäåæçèéêëìíîïðñòóôõöøùúûüþ
0123456789
ΔΔ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Note that using a standard Unicode font (rather than APL385 Unicode used in the table above), the last row above would appear as the circled alphabet, Ⓐ to Ⓩ.

In general:

- Dyalog APL allows for a larger number of characters, at the beginning of and within a legal name, than APL2 does.
- Dyalog APL can handle underbar characters (or their Unicode equivalent) concurrently with lower case characters while APL2 cannot.
- Dyalog does not have name restrictions regarding SΔ and TΔ.
- Dyalog APL names can begin with an underbar while APL2 names cannot.
- In APL2, names can contain a high minus.

Namespace

A namespace is an encapsulation mechanism for identifiers (e.g., names of functions, variables) and their contents so that collisions can be avoided. It provides a method to package portions of code and data. In the APL world, a namespace's contents can co-exist with the active workspace but remain independent of each other.

Namespace concepts and operations are significantly different between the two dialects. Some aspects are listed below.

In Dyalog APL:

- Namespaces are analogous to nested workspaces.
- Most namespaces are saved with the current workspace.
- The session namespace (⎕SE) is a system namespace which is not saved with the current workspace and exists across workspace)LOAD and)CLEAR types of operations.
- A robust set of system commands, user commands, system functions and system variables exist which can be used to manage namespaces.

In APL2:

- Namespaces are independent of any workspace, and stored separately.
- Namespaces are created from saved workspaces using an external tool called PACKAGE, accessed via the ⎕NA system command.
- Namespace contents are accessed using the ⎕NA system command.
- Changes to namespace objects are local to the active workspace and saved with them. if a change is made to the original workspace used for the namespace, and the namespace repackaged, the local objects will be reinitialized from it.
- In order to perform a number of operations in the context of a namespace (see the environment, define new variables, define new functions, etc.), certain cover functions need to be created and saved in each namespace for that purpose.
- APL2 is provided with a number of external functions for namespaces (e.g., PACKAGE, QNS). As external routines, their usage is hard to discover in documentation and thru standard discovery techniques.

In general, namespaces are very complicated to understand and use in APL2. They are much easier to understand and use in Dyalog APL. The one major benefit of those of APL2 is that any workspaces saved using "local" copies of

namespace objects will be manually refreshed if the source external namespace is updated. With Dyalog APL, a copy would have to be done from a saved workspace to have these automatically refreshed (although this does provide some protection for bad updates). This could be automated thru mechanisms, such as a latent expression, for important workspaces.

Shared Variables

APL2 uses shared variables to communicate between two processors where the processor can be an auxiliary processor, an APL user or an APL2 interpreter. When an auxiliary process, often used along with sample workspaces, to support interfaces to external files, external programs, specific hardware, other subsystems and operating system services.

Examples of auxiliary processors used with shared variables include (varies based on environment):

AP Number	Description
100	Host System Command Processor
101	Alternate Input (Stack) Processor
102	Main Storage Access Processor
110	CMS File Processor
111	QSAM File Processor
119	TCP/IP Socket Interface Processor
120	Session Manager Command Processor
121	APL2 Data File Processor
123	VSAM File Processor
124	Text Display Processor
126	GDDM Processor
127	SQL (DB2) Processor
144	X Windows API (Unix only)
145	GUI services processor (Windows only)
200	Calls to APL2 processor
207	Universal Graphics Auxiliary Processor
210	BDAM File Processor for TSO, DOS file management
211	APL2 Object File Processor
227	ODBC processor
488	IEEE GPIB auxiliary processor (Windows only)
499	Sample Auxiliary Processor
<dynamic>	Shared variable interpreter interface

Dyalog APL uses shared variables to communicate with another APL workspace or with another application.

System Limitations

System limitations for APL vary depending on the implementation, platform and product.

The table below shows the limitations imposed on Dyalog APL.

Limitation	Dyalog APL
Largest and smallest representable numbers in an array	1.797693135E308 / ¯1.797693135E308 (depends on ⎕FR)
Most infinitesimal (near 0) representable numbers in an array	2.2250738585072014E¯308 / 4.9406564584124654E¯324 or 1E¯6176 (depends on ⎕FR)
Maximum rank of an array	15
Maximum length of any axis in an array	¯1+2*64
Maximum product of all dimensions in an array	¯1+2*64
Maximum depth of an array applied with the primitive functions depth (⍴R) and match (L≡R)	limited only by memory
Maximum depth of a shared variable	limited only by memory
Maximum depth of a copied variable	limited only by memory
Maximum number of characters in the name of a shared variable	limited by number of lines
Maximum number of characters in a comment (minus leading blanks)	limited by number of lines
Maximum length of line	limited by number of lines
Maximum number of lines in a defined function or operator	4095
Maximum number of labels in a defined function or operator	limited by number of lines
Maximum number of local names (excluding labels) in a defined function or operator	4094
Bytes in internal symbol label	limited only by memory
Maximum number of slots in the internal symbol table. A slot is required for each unique name, each unique constant, and each ill-formed constant in the workspace.	limited only by memory
Value range of ⎕PW	42 to 32767
Maximum value of ⎕PP	34 When ⎕FR is 645, values > 17 are treated as 17.

The table below shows the limitations imposed on APL2.

Limitation	Workstations	APL2/370	APL2/PC	
			16-Bit	32-Bit
Largest and smallest representable numbers in an array	1.7976931348623158E+308 and ¯1.7976931348623158E+308	7.2370055773322621E75 and b7.2370055773322621E75	1.7976931348623158E+308 and ¯1.7976931348623158E+308	1.7976931348623158E+308 and ¯1.7976931348623158E+308
Most infinitesimal (near 0) representable numbers in an array	2.2250738585072014Eb308 and b2.2250738585072014Eb308	5.397605346934027891Eb79 and b5.397605346934027891Eb79	2.2250738585072014Eb308 and b2.2250738585072014Eb308	2.2250738585072014Eb308 and b2.2250738585072014Eb308
Maximum rank of an array	63	64	63	63
Maximum length of any axis in an array	¯1+2*31 (2147483647)	¯1+2*31 (2147483647)	65520	¯1+2*31 (2147483647)
Maximum product of all dimensions in an array	¯1+2*31 (2147483647)	¯1+2*31 (2147483647)	65520	¯1+2*31 (2147483647)
Maximum depth of an array applied with the primitive functions depth (_R) and match (L_R)	181	181	No limit	No limit
Maximum depth of a shared variable	181	181	6	6
Maximum depth of a copied variable	181	181	168	168
Maximum number of characters in the name of a shared variable	255	255	18	18
Maximum number of characters in a comment (minus leading blanks)	4090	32764	4089	4089
Maximum length of line	8190	N/A	4096	4096
Maximum number of lines in a defined function or operator	¯1+2*15 (32767)	¯1+2*31 (2147483647)	700	545
Maximum number of labels in a defined function or operator	Limited by number of lines	32767	700	545
Maximum number of local names (excluding labels) in a defined function or operator	Limited by lengths of lines and names	32767	Limited by lengths of lines and names	Limited by lengths of lines and names
Bytes in internal symbol label	8191	N/A	64999	4095
Maximum number of slots in the internal symbol table. A slot is required for each unique name,	N/A	32767	N/A	N/A

Limitation	Workstations	APL2/370	APL2/PC	
			16-Bit	32-Bit
each unique constant, and each ill-formed constant in the workspace.				
Value range of ⎕PW	30-254	30-390	30-254	30254
Maximum value of ⎕PP	16	18	16	16

Symbol Meaning Differences

For some symbols, APL2 and Dyalog have, or have had different functions associated with them at times due to early adoption or a change in functionality over time.

One example is ∈ which, in the original APL2 IUP had meant Type and Dyalog APL originally did too. By the time IBM released APL2, ∈ was repurposed for Enlist. Dyalog APL left the Type meaning in place only when migration level is 0.

Others include Mix and First, which reverse their meanings between the dialects (adding to the confusion is that they also use different names, mix and disclose, for the same operation.

Symbol	APL2	Dyalog APL
∈	enlist	enlist
↑	first	mix
⊃	disclose	first
⊂	partition	partitioned enclose
⍸		partition

The following table attempts to summarize different meanings of select symbols between the two APL dialects.

Glyph	Discussion
+	Conjugate (+) is a monadic scalar function which negates the imaginary component of a complex argument. APL2 uses a strict conjugate, only applying it to numbers and returning a DOMAIN ERROR in other cases. In Dyalog APL, Conjugate will allow a non-numeric argument (e.g., character) and return it unchanged even though other monadic scalar functions give a DOMAIN ERROR.
^ ∧	Dyalog APL supports (and honors) both ^ (circumflex accent) and ∧ (logical and) while APL2 only supports the first of these. This does not occur in all cases, such as when anchoring a search string with ⎕S. Dyalog APL supports LCM but APL2 does not. LCM (∧) is a dyadic scalar function which returns the Least Common Multiple of two integer arguments. It is an extension of And which maintains the same results on Boolean arguments and the same identity element 1, in the same way that GCD extends Or.
∨	Dyalog APL supports GCD but APL2 does not. GCD (∨) is a dyadic scalar function which returns the Greatest Common Divisor of two integer arguments. It is an extension of Or which maintains the same results on Boolean arguments and the same identity element 0, in the same way that LCM extends And.
≠	Both APL2 and Dyalog APL support this as an unequal comparison, which is a dyadic function using this symbol. Dyalog APL also supports the unrelated monadic function Unique Mask which returns a Boolean result that marks the first occurrence of each element in the argument.

Glyph	Discussion

↑	Both APL2 and Dyalog APL support the dyadic form of this function, Take. Unlike APL2, which will generate a LENGTH ERROR, Dyalog APL will extend the left argument if its length is less than the rank of the right argument. For example:
	
	`      1↑2 3 4⍴⍳24`
	`1 2 3 4`
	`5 6 7 8`
	`9 10 11 12`
	
	The monadic function represented by this symbol differs between APL2 and Dyalog APL.
	
	In APL2, ↑ is First.
	
	In Dyalog APL, ↑ is Mix.

↓	Both APL2 and Dyalog APL support the dyadic form of this function, Drop. Unlike APL2, which will generate a LENGTH ERROR, Dyalog APL will extend the left argument if its length is less than the rank of the right argument. For example:
	
	`      1↓2 3 4⍴⍳24`
	`13 14 15 16`
	`17 18 19 20`
	`21 22 23 24`
	
	Dyalog APL also supports a monadic form of this function, Split. The result is the sub-arrays of the argument along the Kth axis. If an axis is not specified, the last axis is used. For example:
	
	`      ↓2 5⍴⍳10`
	`1 2 3 4 5  6 7 8 9 10`
	`      ↓[1]2 5⍴⍳10`
	`1 6  2 7  3 8  4 9  5 10`

⊃	In APL2, this represents the monadic function Disclose but in Dyalog APL, by default, it represents the monadic function First. For the most part, APL2 and Dyalog APL swapped the functionality of ↑ and ⊃ (Mix and Disclosure basically perform the same functionality).
⊂	In APL2, this represents the dyadic function Partition but in Dyalog APL, by default, it represents the dyadic function Partitioned Enclose (which is not supported in APL2).
⊆	In APL2, this symbol has no functionality but in Dyalog APL, it represents the dyadic function Partition and the monadic function Nest which encloses the argument, if a simple array.
≠	APL2 does not support this. When used monadically, Tally is a primitive monadic function which returns the number of major cells in its argument. When used dyadically, Not Match is a primitive dyadic function which is the negation of Match.
←	There are a number of assignment differences between APL2 and Dyalog APL.

Glyph	Discussion

	Both Dyalog APL and APL2 accept multiple variable assignment as shown here: `(a b)←1 2`. Dyalog APL also supports `a b ←1 2`.
	
	Dyalog APL also supports modified assignment. For the dyadic function `f`, `Xf←Y` is equivalent to `X←XfY`. For example:
	
	`A←1 2 3 4 5`
	`A+←10`
	`A`
	`11 12 13 14 15`
	
	Workstation versions of APL2 may not support all the selective specification capabilities in APL2 Mainframe versions.
	
	Selective specification is not implemented for all cases. For example:
	`(L0¨V)←A`
	produces a `DOMAIN ERROR`.
	
	Any usage of bracket indexing in a selective assignment expression, such as
	`(2↑V[⍳4])←3`
	produces a `SYNTAX ERROR`.
	
	Selective specification of a shared variable also gives `SYNTAX ERROR`.

⌸	APL2 does not support this, but Dyalog APL does. Key (⌸) is a primitive monadic operator which takes a dyadic function operand where specified keys group the indices or major cells of an argument. It is commonly compared to SQL's GROUP BY statement.
⌺	APL2 does not support this, but Dyalog APL does. Stencil (⌺) is a primitive dyadic operator that applies its left operand to (possibly overlapping) rectangular views of the right argument array. The shape and movement of the rectangular views are dictated by the right operand.
⍠	APL2 does not support Variant. Variant (⍠) is a special dyadic operator which takes a function as its left operand and modifies its behavior as specified by the array right operand. Since ⍠ was not in Dyalog APL's atomic vector when the feature was added, a system operator alternative was provided in form of ⎕OPT.

/ ⌿	APL2 and Dyalog APL differ in the behavior of Replicate for negative values.
	
	In APL2, a negative left argument in position I populates the appropriate position in the result with a fill item (using a fill item determined by the type of the first item in the Ith subarray along the last axis of the right argument) the number of times specified by the absolute value of the left argument.
	
	`0 2 ¯3 1 / 1 2 3`
	`2 2 0 0 0 3`

Glyph	Discussion

	In Dyalog APL, a negative left argument populates the appropriate position in the result with a fill (based on the corresponding item in the right argument) item the number of times specified by the absolute value of the left argument.
	
	`      0 2 ¯3 1 / 1 2 3 4`
	`2 2 0 0 0 4`
	
	That is, with APL2, negative numbers in the position of the left argument do not correspond to an element in the same position in the right argument, and the arguments are therefore unequal. With Dyalog APL, negative numbers in the position of the left argument correspond to an element in the same position in the right argument (but use it only to determine the fill number), and the arguments are therefore equal.
	
	Both dialects support left and right scalar extension.
	
	Interestingly, this does not error in APL2 due to a loosening of the conformability requirements:
	
	`      ¯3/2`
	`0 0 0`

/ ≠	In APL2, Replicate is a monadic operator. In Dyalog APL it is a function.
	
	The syntax a / b is ambiguous: it may be an invocation of a dyadic function / with left argument a and right argument b, or of a monadic operator with operand a and right argument b. Note that a function Replicate can be used as an operand while an operator Replicate cannot.
	
	In APL2, to use Replicate as a function, wrap it in a function. For example:
	
	`      ∇r←x R y`
	`[1]   r←x/y`
	`[2]   ∇`
	
	The impact of this characteristic of Replicate can be seen with Replicate Each.
	
	In APL2, Replicate is implemented as an operator. In the example below, it will be applied to the operand 1 3, and Each will be applied to the resulting derived function 1 3/.
	
	`      1 3 /¨ 'ab' 'cd'`
	`abbb  cddd`
	`      (1 3/)¨ 'ab' 'cd'`
	`abbb  cddd`
	
	In Dyalog APL, Replicate is implemented as a function. In the example below, Each will apply to Replicate only, and the resulting derived function will be invoked monadically.
	
	`      1 3 /¨ 'ab' 'cd'`
	`ab  cccddd`
	`      1 3 (/¨) 'ab' 'cd'`

Glyph	Discussion
	ab cccddd
\ ⍀	In APL2, Expand accepts a Boolean while in Dyalog APL, integers are allowed. For example, in Dyalog APL: ρ⎕←1 ¯2 3 ¯4 5 0\'ABC' A BBB CCCCC 16
⍤	The Each operator performs differently between the dialects when an empty argument is passed as one of the arguments (e.g., Z←FOO⍤⍳0, Z←A FOO⍤⍳0, Z←⍳0 FOO⍤A). In APL2, if the argument is empty (for a monadic function) or either argument is empty (for a dyadic function), the operand function is not called. Instead a fill function (which varies based on the operand function passed to Each) is performed. In Dyalog APL, if the argument is empty (for a monadic function) or either argument is empty (for a dyadic function), the operand function is called with the prototype(s) of the argument(s) passed to the function. This difference in behavior can cause differences between the dialects if an empty argument was expected to be passed to the operand function (especially if the operand function was a user-defined function). Z←FOO⍤⍳0
⍪	APL2 does not support this function while Dyalog APL does. When used monadically. Table (⍪) returns a 2-dimensional matrix of the elements of its argument in row-major order, preserving the shape of the first dimension of the argument (if it exists). Table has been implemented according to the Extended APL Standard (ISO/IEC 13751:2001). When used dyadically, Catenate First (⍪) performs catenation along the first (instead of the last) axis.
∪	APL2 does not support this while Dyalog APL supports this as a monadic function and a dyadic function. When used monadically, Unique (∪) returns the unique elements of its array argument (e.g., the unique items of a vector, the unique rows of a matrix), in the order in which they first appear in the argument. When used dyadically, Union (∪) returns the elements of its left vector argument catenated with the elements of its right vector argument which are not found in the left argument.
∩	APL2 does not support this while Dyalog APL supports this as a dyadic function. Intersection (∩) returns a vector composed of items occurring in both its left argument and its right argument in the order of occurrence in the left argument.
⊢	APL2 does not support this while Dyalog APL supports this as a monadic function and a dyadic function. When used monadically, Same (⊢ and ⊣) return the argument, unchanged.

Glyph	Discussion
	When used dyadically, Right (⊢) returns the right argument, unchanged.
⊣	APL2 does not support this while Dyalog APL supports this as a monadic function and a dyadic function. When used monadically, Same (⊢ and ⊣) return the argument, unchanged. When used dyadically, Left (⊣) returns the left argument, unchanged.
?	APL2 returns a DOMAIN ERROR for ?0, while Dyalog APL does not.
⍫	APL2 supports locking a function using Del Tilde (⍫) to start or end an editing session. Dyalog APL instead provides the Lock Definition (⎕LOCK) system function.
∇	In Dyalog APL, when the ∇ symbol is followed by the name of a defined operation and then by a closing ∇, the named operation is displayed. In APL2, the function name must be followed by '[⎕]' to perform this same activity.
∈	By default, the monadic function Enlist (∈) exists in both APL2 and Dyalog APL. Originally Dialog APL had ∈ mean Type, which returned 0 for numeric elements in the argument and space for character. It is now only available when the migration level (⎕ML) is 0, while setting it to 1 (the default) or higher performs the Enlist function.
⌶	APL2 doesn't support I-beam. Dyalog APL supports I-beams as a monadic operator that provides a range of system-related services (and not the I-beam functions of APL\360). There are considered experimental and can change from release to release.
∇	In APL2 and Dyalog APL, Del (∇) it is used to enter and exit the line editor which provides a function definition mode.[1] In Dyalog APL, ∇ also: • represents the current dfn, or the function derived from the current dop, such that anonymous functions and operators may recurse. • is used to delimit tradfn members of scripted objects. • when omitting the function name, causes the suspended operation (that is, the one at the top of the state indicator) to be displayed and opened for editing.
j	APL2 uses 'J' to separate the real and imaginary part of a complex number. Dyalog APL also allows for the use of lower case 'j'.
e	APL2 uses 'E' to separate the multiplier and exponent of a real number entered in Scaled Form, which is also sometimes called the exponential or scientific form. Dyalog APL also allows for the use of lower case 'e '.
D	APL2 supports specifying a complex number in polar notation, by using magnitude and angle in degrees separated by the letter D and no spaces. The number is real if the angle is an integral multiple of 180.
R	APL2 supports specifying a complex number in polar notation, by using magnitude and angle in radians separated by the letter R and no spaces. The number is real if the angle is an integral multiple of pi (π).

Glyph	Discussion
⍬	In Dyalog APL, the empty vector (⍳0) may be represented by the numeric constant Zilde (⍬). APL2 does not support this.
⊃	The function represented by the right shoe symbol (⊃) differs between APL2 and Dyalog when monadic: • In APL2, ⊃ is Mix when monadic. • In Dyalog APL, ⊃ is First when monadic.
⍳	Both APL2 and Dyalog APL support Index Generator. Dyalog APL also supports Index Generator (Indices), a monadic function extension in which, when a vector is passed, the shape of the result is that of the argument and each element of the result occurs in its self-indexing position in the result. In particular, the following identity holds: ⍳Y ←→ (⍳Y)[⍳Y] For example: ⍳2 3 1 1 1 2 1 3 2 1 2 2 2 3 Dyalog APL also supports the following, while APL2 returns a LENGTH ERROR for its equivalent (e.g., ⍳⍳0): ⍳⍬ (⊂⍬) ≡ ⍳⍬ 1 Dyalog APL applies leading-axis rank matching to the argument for dyadic ⍳ where possible.
⍸	APL2 does not support this by Dyalog APL supports both a monadic and dyadic function using this symbol. When used monadically, Where (⍸) returns each index the corresponding number of times specified in the argument (which is an array or non-negative integers). When the argument is Boolean this has the effect of identifying each index that has a value of 1. For example: ⍸0 1 0 0 0 1 2 6 ⍸1 3 2 4 1 2 2 2 3 3 4 4 4 4 When used dyadically, Interval; Index (⍸) identifies in which interval, identified in the left argument, each element of the right argument occurs. The left argument is an ordered non-scalar array (numeric or character) that represents a set of intervals or ranges. In addition to the intervals between each element in the left argument, there is a 0 interval

Glyph	Discussion

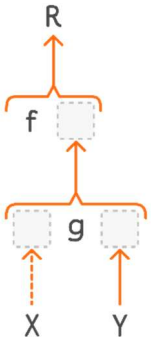
	for those values less than the first interval and an interval for those values greater than the last element. For example:
	
	`      10 20 30 ⍸ 11 1 31 21`
	`1 0 3 2`
	`      'AEIOU' ⍸ 'EXAMPLE1'`
	`2 5 1 3 4 3 2 0`

,	When both arguments to Catenate are empty, the result must also be empty. In this case its shape can be computed without any irregularity, but its prototype is not determined by the definition of Catenate with a non-empty result. In that case, the prototype is obtained from the first result element, but for an empty result there is no such element. As a result, it must be specified: various APLs choose either the left or right argument's prototype. Dyalog APL uses the left argument while APL2 uses the right.

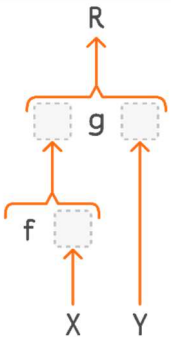
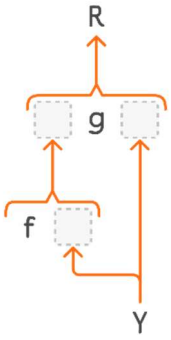
⍕	When monadic format is applied to a nested array, the spacing of the result is sometimes different. For example (using ∘ to represent blanks in the results here for display purposes):
	
	`      ⎕←⍕'FOUND' 9 'DOCUMENTS'`
	`∘FOUND∘∘9∘∘DOCUMENTS∘     ⍝ DYALOG APL`
	`∘FOUND∘9∘DOCUMENTS∘       ⍝ APL2`
	
	There are also subtle differences when some more complex nested arrays are formatted.
	
	Both APL2 and Dyalog APL support the dyadic function of format, Format-by specification, in which the left argument are pairs of numbers specifying width and precision for formatting the right argument.
	
	APL2, but not Dyalog APL, supports a second format of the dyadic function of format, Format-by-example, in which the left argument is a set of patterned character strings where each pattern shows how to format numbers in the right argument.

⌷	Both APL2 and Dyalog APL support Index, which is a dyadic function using this symbol. Dyalog APL also supports the unrelated monadic function Materialise using this symbol. It is related to classes and casts the argument to an array.
≡	By default, Dyalog APL returns negative numbers when the depth is uneven. APL2 returns the maximum depth.
@	Dyalog APL supports the primitive operator At (@) which is not supported by APL2. This operator substitutes selected items in its right argument with new values or applies a function to modify selected items in that argument. A call to At is of the form `X (f @ g) Y`, where X is optional. `g`, which can be a Function or an Array, specifies the indices of the right argument that should be modified. `f`, which can be a Function or an Array, specifies what happens to the elements at those positions. If f is a dyadic function, its left argument is the array `X`.

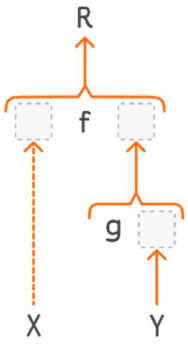
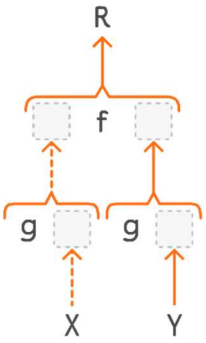
Glyph	Discussion

⍤	In Dyalog APL the symbol Jot Diaresis (⍤) supports multiple primitive operators, whose operations are dependent on the context of its use. These are not supported by APL2.
	
	When the right operand is a numeric value, ⍤ represents the Rank operator, which has both a monadic and dyadic form.
	
	When the right operand is a function, ⍤ represents the primitive operator Atop, which has both a monadic and dyadic form.
	
	The Rank operator applies the left operand successively to sub-arrays of the right argument (when used monadically) or between sub-arrays of the left and right argument (when used dyadically) based on the numeric array information passed in the right operand. or example:
	
	`      10 20 30 (+⍤0 1)3 4⍴⍳12`
	`10 11 12 13`
	`24 25 26 27`
	`38 39 40 41`
	
	The Atop operator allows functions to be glued together to build up more complex functions.
	
	When called dyadically with function operands `f` and `g`, `X f⍤g Y` is equivalent to `f X g Y`.
	
	When called monadically, `f⍤g Y` is equivalent to `fgY`.
	
	 

⍥	Dyalog APL supports the primitive operator Behind (⍥), which is not supported by APL2. It has both a monadic and dyadic form. The Behind operator allows functions to be glued together to build up more complex functions. When called dyadically with function operands `f` and `g`, `X f⍥g Y` is equivalent to `(f X) g Y`.

Glyph	Discussion
	 When called monadically, `f∘g Y` is equivalent to `(f Y) g Y`. 

∘	In APL2, Jot (∘) is only allowed in the context of an Outer Product and is documented that jot "is treated syntactically as a function."[10]
	
	In Dyalog APL the symbol also supports multiple primitive operators, whose operations are dependent on the context of its use. These are not supported by APL2.
	
	When the left or right operand is a numeric value, ∘ represents the Bind operator. which binds an array to a dyadic function. This is described as either left argument currying or right argument currying, based on where the array is. For example:
	
	`SINE ← 1∘○`
	`SINE 10 20 30`
	`¯0.5440211109 0.9129452507 ¯0.9880316241`
	
	`SQRT ← *∘.5`
	`SQRT 4 16 25`
	`2 4 5`
	
	When the left and right operands are functions, ∘ represents the Beside operator, previously named compose, which allows functions to be glued together to build up more complex functions.
	
	When called dyadically with function operands `f` and `g`, `Xf∘gY` is equivalent to `XfgY`.
	
	When called monadically, `f∘gY` is equivalent to `fgY`.

Glyph	Discussion
	
⍤	Dyalog APL supports the primitive operator Over (⍤) composition, which is not supported by APL2. The Over operator allows functions to be glued together to build up more complex functions. When called dyadically with function operands `f` and `g`, `X f⍤g Y` is equivalent to `(gX) f (gY)`. When called monadically, `f⍤g Y` is equivalent to `f g Y`. 
⍨	APL2 does not support this, but Dyalog APL does in two different ways based on the arguments to the operator. When the left operand is an array, Constant always returns the operand. When the left operand is a dyadic function, Commute performs the function with the left and right arguments swapped. If there is no left argument, the right argument is also used ('duplicated") as the left argument to the function.
⍣	APL2 does not support this but Dyalog APL does. The Power dyadic operator repeated calls a function either a specified number of times or until a predicate is satisfied, depending on if the right operand is a numeric integer scalar, or a scalar-returning dyadic function, respectively.
{ }	Brackets have a role in Dyalog APL. When used within the header of a traditional function, they are used to identify an optional left argument as well as to identify a shy result. They are also used to identify the specification of a dfns or dops operation.

Glyph	Discussion
⍺	In Dyalog APL, ⍺ represents the left argument in a dfns or a dops.
⍵	In Dyalog APL, ⍵ represents the left argument in a dfns or a dops.
⍺⍺	In Dyalog APL, ⍺⍺ represents the left operand in a dops.
⍵⍵	In Dyalog APL, ⍵⍵ represents the left operand in a dops.
∇	In Dyalog APL, ∇ is used to specify a recursive call in a dfns or a dops. While the dfns or dops name can be used in named definitions, this symbol allows for the use in unnamed situations (and incurs less interpretative overhead).
∇∇	In Dyalog APL, ∇ is used to specify a recursive call in a dops can also be specified using ∇∇.
:	In addition to being used to identify labels, a colon has additional roles in Dyalog APL. In dfns and dops, it represents a Guard and is of the form `lhs : rhs`. If the left-hand side (lhs) of a guard is 1 then the right-hand side (rhs) is evaluated as the result of the function. In Tradfns, it represents the beginning of a control structure.
::	In Dyalog APL, an error-guard can be used to trap error conditions in a dfns or dops. It is of the `error number(s):: expression`. If during operation of the dfns or dops an error occurs, and the error number is specified in an error-guard, the associated expression is evaluated. There can be multiple error-guards defined for the same error number allowing for cascading expressions to be evaluated.
#	In Dyalog APL, this is used to identify the top level or 'Root' namespace. ## is the parent or space containing the current namespace.
##	In Dyalog APL, this is used to identify the parent or space containing the current namespace.
.	Both APL2 and Dyalog APL support the use of a as a decimal separator (`3.14`), for Inner product and for Outer product. Dyalog APL also recognizes it for two other roles: - as part of a specification for a namespace name (e.g., of the form namespace.name) and - when reference array expansion is used in certain situations (e.g., namespaces, .NET classes)
&	Dyalog APL supports the monadic primitive operator Spawn (&), which is not supported by APL2. It spawns a new thread in a function is run against provided arguments.

Mix / Disclose

The Mix / Disclose monadic function is supported by both APL2 (⊃) and Dyalog APL (↑) but using different symbols.

In APL2 disclose requires elements to have the same rank but pads them to a common shape, while in Dyalog APL, mix allows elements with different ranks to be combined.

Both APL2 and Dyalog APL support specifying an optional axes value. Each supports the axes being an integer vector, with the same length as the shape (⍴) of the argument, each element of which specifies the position in the result of the corresponding axis of the argument.

Dyalog APL supports two additional types of axes specification:

- The axes may be a scalar, or 1-element vector whose value is a fractional number indicating the two axes of the argument between which new axes are to be inserted for the argument. The shape of the result is the shape of the argument with the shape or the argument inserted between the axes of the argument represented by the floor and the ceiling of the axis specification.

- The axis may be a scalar, or 1-element vector integer, whose value specifies the position of the first axis of the argument in the result. This case is identical to the fractional case where the axes specified (in this case) is the same as the ceiling of the axes (as would have been specified in the fractional case).

Scalar Extension

APL2 only extends a one-element vector and not any singleton (an array with one element). Dyalog APL also extends singletons, discarding its shape. The following generates a RANK ERROR in APL2 but is accepted in Dyalog APL:

```
      (1 1⍴5) + 10 20
15 25
```

Inner Product

The dialects perform alike when the operands of Inner Product are scalar functions.

There are differences between the dialects when other arguments are used.

The most common form is that APL2 breaks the left argument into rows (vectors along the last axis) and the right argument into columns (vectors along the first axis) and performs a vector-vector inner product along each possible pair.

With Dyalog APL the operands are applied strictly to individual elements.

In particular, given vectors A and B:

```
A f.g B  ←→  ⊃f/A g B  ⍝ APL2
A f.g B  ←→  f/A g¨ B  ⍝ Dyalog APL
```

For example:

APL String	APL2 Result	Dyalog APL Result
`1 2 3 ,., 4 5 6`	`1 2 3 4 5 6`	`1 4 2 5 3 6`
`≡1 2 3 ,., 4 5 6`	3	2

`2 3 +.ρ 4 5`	`13 14`	`LENGTH ERROR: Mismatched left and right argument shapes`
		`2 3+.ρ4 5`
		`^`

`≡2 3 +.ρ 4 5`	2	
`3 3 +.ρ 4 5`	`13 14 13`	`9 9 9`
`≡3 3 +.ρ 4 5`	2	2

In addition, as shown above, the depth of the results may be different between APL2 and Dyalog APL.

Internal Data Types

APL2 supports the following internal data types:

Boolean
integer
real
complex
byte integer
short character (1 byte per character; ⎕AV based)
long character (4 bytes per character; 2 bytes contain the data's codepage and 2 bytes contain the data's code point)
array (i.e. not a simple scalar)

Dyalog APL supports the following internal data types (as returned by the system function ⎕DR – Data Representation):

1 bit Boolean
8 bits character
8 bits signed integer
16 bits character
16 bits signed integer
32 bits character
32 bits signed integer
Pointer (32-bit or 64-bit as appropriate)
64 bits Floating
128 bits Decimal
128 bits Complex

Numeric types

APL2 and Dyalog APL both support real numbers, numbers specified in scaled (E) format, and complex numbers specified using 'J' format. Dyalog APL allows E and J to be entered in lower case, while APL2 does not.

APL2 also supports entering a complex number on degree and radian formats, which is not available in Dyalog APL.

In Dyalog 2, the numeric constant Zilde (⍬) represents the empty vector (⍳0).

Dyalog APL uses 64-bit float for most numbers. It supports 128-bit decimal float, allowing for larger number ranges and more precision, by setting the system variable ⎕FR (Floating-Point Representation) to 1287 (for IEEE 754-2008 128-bit decimal floating-point operations) instead of 645 (for IEEE 754 64-bit floating-point operations. Changing the value is not retroactive and only effects calculations and numbers stored after that point.

User-Defined Functions and Operators

Definition

Both Dyalog APL and APL2 support a line editor for defining Functions and Operators. Editing commands and behavior may vary in each.

For example,

APL2:
- supports nested editing sessions while Dyalog APL does not
- displays a saved timestamp when the full function is displayed

Dyalog APL:
- displays the most recent suspended function, and remains in edit mode, when the ∇ symbol is entered on its own
- displays a function when the ∇ symbol is followed by the name of a defined operation and then by a closing ∇

Dyalog APL and APL2 both support additional editors

> APL2 supports an APL full-screen (or windowed) editor, an interface to system editors and an interface to user-written APL editors. The availability of these editors may be platform dependent.

Established and viewed using system functions

> In APL2, Tradfns may be created from a character matrix or vector containing source code using the system function ⎕FX, or from a character vector using the system function ⎕TF. The system functions ⎕CR and ⎕TF return the original source code.
>
> In Dyalog APL, Tradfns may be created from an array containing source code using the system function ⎕FX. The system functions ⎕CR, ⎕VR, ⎕NR return the original source code.

Content

APL2 and Dyalog APL both support Traditionally Defined Functions and Operators (Tradfns) which have function headers.

Dyalog APL also supports another style of Defined Functions and Operators identified as dfns and dops. These are terms unique to Dyalog APL and refer to a special type of function or operator. They were originally named dynamic functions and dynamic operators, later abbreviated to Dfns and Dops or D-Fns and D-Ops, but all these terms have been dropped in favour of the current ones. Wikipedia associates slightly different definitions for these abbreviations (Direct Function) so be careful as there may be differences between the two.

Dyalog also supports a third type of defined programs and operators called tactic functions. A Tactit function is one which does not explicitly identify its argument.

Traditional Defined (e.g., Del) functions

There are differences between APL2 and Dyalog APL in the definitions associated with Traditional Defined Functions.

For example, APL2 Mainframe and Dyalog APL allow comments on line 0, while workstation implementations of APL2 will strip out the comment.

Other major differences are discussed below.

Ambivalent Functions

An ambivalent function is one with an optional left argument. In Dyalog APL, the header of a function must explicitly specify that a function in ambivalent by enclosing the left argument name in curly braces (for example, `result←{left} function right`). This specification is not used in APL2.

Shyness

In Dyalog APL, automatic display of the explicit results of a tradfn can be suppressed by putting curly braces around the result name in the function definition.

For example:

```
      ∇ {res}←left Add right
        res←left+right
      ∇
      1 Add 2       ⍝ result is "shy"
      ⎕←1 Add 2     ⍝ coerce display of result
3
      10×1 Add 2    ⍝ use result anyway
30
```

Namelists

In Dyalog APL, the right argument and / or the result of a function can be a blank-delimited list of names surrounded by parentheses (i.e., Namelist) instead of single name.

When a function has a Namelist for a right argument, the elements of the argument are assigned to the names in the Namelist in the specified order. If the length of the elements passed in the right argument of the calling statement is not the same as the number of names in the Namelist, a `LENGTH ERROR` will result.

For example:

```
      ∇ (c b a)←Reverse(a b c)
      ∇
      Reverse 1 2 3
3 2 1
```

Control Structures

Dyalog APL supports the use of control structure using flow control keywords, which are preceded by a colon. Keywords include `If`, `While`, `Repeat`, `For` (with the supplementary control words `In` and `InEach`), `Select`, `With`, `Trap`, `Hold` and `Disposable`). This is not supported in APL2.

Additional control words are also available:

- `For` control word supplementary control words (e.g., `In` and `InEach`),
- Within a control structure (e.g., `Else`, `ElseIf`, `AndIf`, `OrIf`, `Until`, `Case` and `CaseList`),
- To identify the end of a particular control structure (e.g., `EndIf`, `EndWhile`, `EndRepeat`, `EndFor`, `EndSelect`, `EndWith`, `EndTrap`, `EndHold` and `EndDisposable`; all may all be abbreviated to `End`), and
- To conditionally alter the flow of execution within a control structure (e.g., `GoTo`, `Return`, `Leave` and `Continue`).

For example, a function implementing the Trabb Pardo–Knuth algorithm using control structures:

```
      ∇ {res}←trabb;f;S;i;a;y
        f←{(0.5*⍨|⍵)+5×⍵*3}
        S←,⍎{⎕←⍵ ⋄ (≠⍵)↓⎕}'Please, enter 11 numbers: '
        :For i a :InEach (⍳≠S)(⊂S)
            :If 400<y←f(a)
                ⎕←'Too large: ',⍕i
            :Else
                ⎕←i,y
            :EndIf
        :EndFor
      ∇
```

Dynamic localization

In Dyalog APL, names can be localized dynamically, while a tradfn is running, using the Shadow Name (`⎕SHADOW`) system function.

```
      name←1 2 3
      ∇ res←Dummy
        ⎕SHADOW'name'
        name←42
        res←name
      ∇
      Dummy
42
      name
1 2 3
```

Locals name lines

In Dyalog APL, additional local names can also be declared on separate lines following the header line:

```
      name←1 2 3
      ∇ res←Dummy
        ;name
        name←42
        res←name
      ∇
      Dummy
42
      name
1 2 3
```

Function Declaration Statements

Function Declaration statements are used to identify the characteristics of a function in some way.

Currently these are all methods related:

- `:Access`
- `:Attribute`
- `:Implements`
- `:Signature`

Dfns and Dops

Dfns and dops represent Dyalog APL's variant of direct definition. Some characteristics include:

- they do not have a function header,
- they can be immediately executed (e.g., in command line or within another function) without assignment to a name
- they can be assigned to a name using a right arrow
- names assigned in the function are localized
- the result is implicit
- execution terminates, for multiline definitions, on the first line that is not an assignment.

A dfns or dops is an APL expression enclosed in curly braces {}, possibly including the special characters ⍺ and ⍵ to represent the left and right arguments of the function respectively. If the presence of either of the compound symbols ⍺⍺ or ⍵⍵ anywhere in its definition, the operation is considered a dops. These may be created as single line or as multi-line operations. They may, or may not, be assigned to a name.

For example:

```
      {(+/⍵)÷⍴⍵} 1 2 3 4     ⍝ Arithmetic Mean (Average)
2.5
      3 {⍵*÷⍺} 64            ⍝ ⍺th root
4
      mean←{(+/⍵)÷⍴⍵}
      mean 1 2 3 4
```

2.5

Several of the same capabilities available with traditional functions (e.g., shyness) can also be achieved with dfns and dops.

For complete details regarding dfns and dops, see Dyalog APL documentation.

Tactit Functions

Tactit functions are functions which do not explicitly refer to their arguments (their arguments are implied). For example:

```
      Plus ← +
      IndicesTo ← ⍳
      _Reduce ← /
      Sum ← Plus _Reduce
      Sum IndicesTo 10
```

Note: dfns are not tactic functions (although at least one Dyalog APL training example says it is). For example, the following is a dfns, `iota←{⍳⍵}`, while the similar tactic function is `iota←⍳`.

However, dfns can be used in the definition of a tactic function and the result is a tactic function. For example, given:

```
iota←{⍳⍵}
trim←{7↑⍵}
```

the following is a dfns and not a tactic function:

```
max7iota←{trim iota ⍵}
```

while the following is a tactic function:

```
max7iota←trim iota
```

Tactit Functions as a Traditional Function Result

A tradfn can return a tactit function as result. The returned function will replace the function and its arguments (if any) in the calling expression. For example,

```
      ∇ Fn←Apply name
        :If name≡'plus'
            Fn←+
        :ElseIf name≡'times'
            Fn←×
        :EndIf
      ∇
      3(Apply'plus')4
7
      3(Apply'times')4
12
```

Trains

Dyalog APL supports trains. A Train is a derived function constructed from a sequence of 2 or 3 functions, or from an array followed by two functions, which bind together to form a function. As a train resolves to a function, a sequence of more than 3 functions represents a train of trains. While powerful, it makes reading code hard (using `]boxing` help). For example:

```
      6( +,-,×,÷)2      ⍝ (6+2),((6-2),((6×2),(6÷2)))
8 4 12 3
```

Object Programming

Dyalog APL supports object programming with classes, constructors, methods, etc.

Parallel Execution

APL2 and Dyalog APL provide significantly different mechanisms associated with parallel execution.

APL2 supports Cooperative Processing by providing the ability to distribute data and processing among multiple APL2 systems connected through TCP/IP. The APL2 systems on CMS, TSO, AIX, Linux, Sun Solaris, and Windows all contain the following features:

- Cross system shared variables for synchronization and data transfer in a distributed processing environment
- A TCP/IP socket interface for communicating with non-APL applications (AP 119)
- Program control of an APL2 session by another application or APL2 user
- Remote session manager capability (AP 120)

APL2 also supports parallel processing by calling APL2 from APL2. This can be accomplished by using auxiliary processor AP 200, which provides an asynchronous interface (the external function APL2PIA can be used to call APL2 but provides a synchronous interface and is therefore not parallel). It can be done on the current machine or another machine.

APL2 also provides two external operators, PEACHT (Parallel Each using Threads) and PEACHP (Parallel Each using Processors) to apply a function to individual argument elements in separate asynchronous threads.

PEACHP uses one or more instances of AP 200 to start the asynchronous APL2 interpreters. The interpreters run in threads in the auxiliary processors' processes. PEACHT starts the asynchronous APL2 interpreters in separate threads in the calling interpreter's process. The two operators are similar, except that:

- PEACHP supports distributing work to other machines; PEACHT does not.
- PEACHT supports sharing of resources in the calling interpreter's process; PEACHP does not.
- PEACHP supports simple homogeneous arguments; PEACHT does not. For PEACHT, each argument must be nonhomogeneous or have depth greater than one, unless it has exactly one element.

Some interactions with facilities external to APL2 (available through ⎕CMD or auxiliary processors) may be asynchronous and run in parallel with APL2, but these do not represent parallel operation of APL code.

Dyalog APL supports parallel execution in a number of ways.

Depending on the availability of a multi-CPU computer or a multi-core processor, certain operations (e.g., scalar dyadic ÷, ≥, =, ≤, ⊛, |, !, ○, ⍲ and ∧) may be processed using different system threads.

Dyalog APL also supports programmatic parallel execution thru a number of language elements, including:

- The Primitive operator Spawn (&) which runs a function in a new thread.
- A number of system functions to manage threads: ⎕TID, ⎕TCNUMS, ⎕TNUMS, ⎕TKILL, ⎕TSYNC.
- An extension to the GUI Event syntax to allow asynchronous callbacks.
- A control structure, :Hold, to synchronize resource access between threads.
- System commands)HOLDS (to display held tokens) and)TID (to switch current thread identity).
- System commands)SI and)SINL have been extended to show threads' tree-like calling structure.

Dyalog APL also supports a special kind of namespace known as an isolate inside which all APL language statements are executed in parallel with the main APL process. A workspace (ws\isolate) is distributed with Dyalog APL that includes code showing how it can be used.

Like APL2, some interactions with facilities external to Dyalog APL (e.g., available through ⎕CMD, ⎕SH, .NET, Conga, externally named C functions, Ride and auxiliary processors) may be asynchronous and run in parallel with Dyalog APL, but these do not represent parallel operation of APL code.

Error Handling and Diagnosis Differences

There are several differences between the two dialects in terms of errors and program problem diagnosis.

The Error

For APL code, the single-line (for commands) and three-line (for line-code, functions and operators) error response differ in a number of ways between the dialects (and can even differ within APL2 implementations).

For commands, the message often differs in content and case. For example:

Command	APL2 Error	Dyalog APL Error
)LIB *APL	INCORRECT COMMAND	FILE ACCESS ERROR: *APL: Unable to scan directory ("The filename, directory name, or volume label syntax is incorrect.")
)OPOP	INCORRECT COMMAND	Incorrect command
)LOAD TEST	NOT FOUND	ws not found
)SI L	INCORRECT COMMAND	invalid parameter

For line code, functions and operators, the first line associated with an error, which represents a line of code, can differ because APL code is parsed and converted to internal form. Upon the occurrence of an error, the code which is displayed is re-generated from the parsed code or internal format. As such, it can be different.

The second line of code represents the error message itself. Like command error messages, code messages often differ in content and case. In addition, Dyalog APL messages are sometimes followed by a colon and additional error details. For example:

Command	APL2 Error	Dyalog APL Error
2<	SYNTAX ERROR	SYNTAX ERROR: Missing right argument

A significant bonus in APL2 is the AXIS ERROR, which doesn't exist in Dyalog. Dyalog uses DOMAIN ERROR, RANK ERROR or SYNTAX ERROR.

Command	APL2 Error	Dyalog APL Error
⍳[1.5]1	AXIS ERROR	SYNTAX ERROR

Major errors for each dialect are shown in the tables below. Each dialect has differences in error numbers for errors.

Error #	APL2 Error Description
0 0	No error
0 1	Unclassified event
1 1	INTERRUPT
1 2	SYSTEM ERROR
1 3	WS FULL
1 4	SYSTEM LIMIT - symbol table
1 5	SYSTEM LIMIT - interface unavailable
1 6	SYSTEM LIMIT - interface quota
1 7	SYSTEM LIMIT - interface capacity
1 8	SYSTEM LIMIT - array rank
1 9	SYSTEM LIMIT - array size
1 10	SYSTEM LIMIT - array or internal function depth
1 11	SYSTEM LIMIT - prompt length
1 12	SYSTEM LIMIT - value unrepresentable
1 13	SYSTEM LIMIT - implementation restriction
2 1	Required operand or right argument omitted
2 2	Ill-formed line
2 3	Name class
2 4	Invalid operation in context
2 5	Compatibility setting prohibits this syntax
3 1	Name with no value
3 2	Function with no result
4 1	⎕PP ERROR
4 2	⎕IO ERROR
4 3	⎕CT ERROR
4 4	⎕FC ERROR
4 5	⎕RL ERROR
4 7	⎕PR ERROR
5 1	VALENCE ERROR
5 2	RANK ERROR
5 3	LENGTH ERROR
5 4	DOMAIN ERROR
5 5	INDEX ERROR
5 6	AXIS ERROR

Error #	APL2 Error Description
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted function call
5	Input/output error
6	No such device or address
7	Arg list too long
8	Exec format error
9	Bad file descriptor
10	No child processes
11	Resource temporarily unavailable
12	Not enough space
13	Permission denied
14	Bad address
15	Unknown error
16	Resource device
17	File exists
18	Improper link
19	No such device
20	Not a directory
21	Is a directory
22	Invalid argument
23	Too many open files in system
24	Too many open files
25	Inappropriate I/O control operation
26	Unknown error
27	File too large
28	No space left on device
29	Invalid seek
30	Read-only file system
31	Too many links
32	Broken pipe
33	Domain error
34	Result too large
35	Unknown error
36	Resource deadlock avoided
37	Unknown error
38	Filename too long
39	No locks available
40	Function not implemented
41	Directory not empty
42	Illegal byte sequence
43	Unknown error

Table 1 - Sample APL2 Errors

Error #	Dyalog APL Error Description
1	WS FULL
2	SYNTAX ERROR
3	INDEX ERROR
4	RANK ERROR
5	LENGTH ERROR
6	VALUE ERROR
7	FORMAT ERROR
10	LIMIT ERROR
11	DOMAIN ERROR
12	HOLD ERROR
13	OPTION ERROR
15	LST FULL
16	NONCE ERROR
17	ACCESS ERROR
18	FILE TIE ERROR
19	FILE ACCESS ERROR
20	FILE INDEX ERROR
21	FILE FULL
22	FILE NAME ERROR
23	FILE DAMAGED
24	FILE TIED
25	FILE TIED REMOTELY
26	FILE SYSTEM ERROR
28	FILE SYSTEM NOT AVAILABLE
30	FILE SYSTEM TIES USED UP
31	FILE TIE QUOTA USED UP
32	FILE NAME QUOTA USED UP
34	FILE SYSTEM NO SPACE
35	FILE ACCESS ERROR - CONVERTING FILE
36	INCOMPATIBLE ARRAY
38	FILE COMPONENT DAMAGED
52	FIELD CONTENTS RANK ERROR
53	FIELD CONTENTS TOO MANY COLUMNS
54	FIELD POSITION ERROR
55	FIELD SIZE ERROR
56	FIELD CONTENTS/TYPE MISMATCH
57	FIELD TYPE/BEHAVIOUR UNRECOGNISED
58	FIELD ATTRIBUTES RANK ERROR
59	FIELD ATTRIBUTES LENGTH ERROR
60	FULL-SCREEN ERROR
61	KEY CODE UNRECOGNISED
62	KEY CODE RANK ERROR
63	KEY CODE TYPE ERROR

Error #	Dyalog APL Error Description
70	FORMAT FILE ACCESS ERROR
71	FORMAT FILE ERROR
72	NO PIPES
73	LOST SLAVE
76	UNABLE TO FORK PROCESS
77	[AP EXIT]
84	TRAP ERROR
86	FUTURE ERROR
90	EXCEPTION
91	EXTERNAL DLL EXCEPTION
92	TRANSLATION ERROR
93	UNACCEPTABLE CHARACTER
94	CHARACTER NOT REPRESENTABLE IN CLASSIC
98	AFFIRM ERROR
99	INTERNAL ERROR
101	FILE ERROR 1 Operation not permitted
102	FILE ERROR 2 No such file or directory
103	FILE ERROR 3 No such process
104	FILE ERROR 4 Interrupted function call
105	FILE ERROR 5 Input/output error
106	FILE ERROR 6 No such device or address
107	FILE ERROR 7 Arg list too long
108	FILE ERROR 8 Exec format error
109	FILE ERROR 9 Bad file descriptor
110	FILE ERROR 10 No child processes
111	FILE ERROR 11 Resource temporarily unavailable
112	FILE ERROR 12 Not enough space
113	FILE ERROR 13 Permission denied
114	FILE ERROR 14 Bad address
115	FILE ERROR 15 Unknown error
116	FILE ERROR 16 Resource device
117	FILE ERROR 17 File exists
118	FILE ERROR 18 Improper link
119	FILE ERROR 19 No such device
120	FILE ERROR 20 Not a directory
121	FILE ERROR 21 Is a directory
122	FILE ERROR 22 Invalid argument
123	FILE ERROR 23 Too many open files in system
124	FILE ERROR 24 Too many open files

Error #	Dyalog APL Error Description
125	FILE ERROR 25 Inappropriate I/O control operation
126	FILE ERROR 26 Unknown error
127	FILE ERROR 27 File too large
128	FILE ERROR 28 No space left on device
129	FILE ERROR 29 Invalid seek
130	FILE ERROR 30 Read-only file system
131	FILE ERROR 31 Too many links
132	FILE ERROR 32 Broken pipe
133	FILE ERROR 33 Domain error
134	FILE ERROR 34 Result too large
135	FILE ERROR 35 Unknown error
136	FILE ERROR 36 Resource deadlock avoided
137	FILE ERROR 37 Unknown error
138	FILE ERROR 38 Filename too long
139	FILE ERROR 39 No locks available
140	FILE ERROR 40 Function not implemented
141	FILE ERROR 41 Directory not empty
142	FILE ERROR 42 Illegal byte sequence
1001	Stop vector
1002	Weak interrupt
1003	INTERRUPT [strong]
1004	INPUT INTERRUPT
1005	EOF INTERRUPT
1006	TIMEOUT
1007	RESIZE
1008	DEADLOCK

Table 2 - Sample Dyalog APL Errors

In APL2, the third line of an error contains two carets pointing to the expression. The left caret indicates how far APL2 interpreted the statement—from right to left. The right caret indicates where the error or interrupt occurred.

Dyalog APL only supports one caret, identifying where the error or interrupt occurred.

For example:

APL2 Error	Dyalog APL Error
LENGTH ERROR 2 1↑1 2 3 ^ ^	RANK ERROR 2 1↑1 2 3 ^

Error reporting and programmatic creation of errors exists in both dialects, While both capture error messages in system variables and allow for the creation of an error, the associated mechanisms are different in each.

For example (this is not a exhaustive display):

Description	APL2	Dyalog APL
Command	2x	2x
Error	SYNTAX ERROR 2× ^^	VALUE ERROR: Undefined name: x 2 x ^
Error Message	⎕EM SYNTAX ERROR 2× ^^	⎕dm VALUE ERROR 2 x ^
Error / Event Number	⎕ET 2 1	⎕en 6
Extended Diagnostic Message		⎕dmx EM VALUE ERROR Message Undefined name: x ⎕em ⎕en VALUE ERROR

Dyalog APL also supports ⎕EXCEPTION, which reports the most recent Microsoft .NET Exception.

Error Trapping

In APL2, error trapping is only available for specified code, and not as a general handling for sections of code in a function. This is done through two system functions (⎕EA and ⎕EC) that allow for the handling of errors.

With Execute Alternate (⎕EA), if the right argument returns an error (or no result), the left argument is executed.

With Execute Controlled (⎕EC), the right argument is executed and the result includes a three-item value with the first position characterizing the result of executing the right argument, the second item the ⎕ET value associated with the result, and the third item being the result. The characterization values and sample string that would generate each are:

0 Error (2+'A')
1 Expression with a result which would display (2+3)
2 Expression with a result which would not display (A←2+3)
3 Expression with no explicit result (F X where F has no result)
4 Branch to a line (→3)
5 Branch escape (→)

Dyalog APL allows for establishing error handling for specific conditions through two mechanisms.

This first is the trap system function (⎕TRAP) which allows for establishing what specific actions that are to be taken for specified error numbers. For example, the following will trap for a DOMAIN ERROR:

```
      ⎕TRAP←11 'E' '''BAD'''
      1÷0
BAD
```

The second mechanism for trapping occurs with a function using the :TRAP control statement, which specifies one or more error codes which are to be trapped in a function. Lines in the function are executed until a :ELSE, :CASE or :CASELIST, or :END[TRAP] statement is observed. If no error occurs, execution continues with the code following the :END[TRAP] statement. If an error occurs which is in the list specified on the :TRAP statement, any associated :CASE, :CASELIST statement code is executed (if one includes a matching error code), or the code associated with the :ELSE statement (for any other codes, if an :ELSE statement is provided). Execution then occurs following the associated :END[TRAP] statement. The following is an example code segment associated with APL file access:

```
[1]  :Trap 17+⍳21 ⍝ Component file errors.
[2]     tie←name ⎕FTIE 0 ⍝ Try to tie file
[3]    'OK'
[4]  :Case 22
[5]     'Can''t find ',name
[6]  :CaseList 25+⍳13
[7]     'Resource Problem'
[8]  :Else
[9]     'Unexpected Problem'
[10] :EndTrap
```

Error Generation

Both APL2 and Dialog APL can generate an error in a manner similar to native systems errors.

In APL2, this is done with the Event Simulate for system function (⎕ES). It can be used in several ways:

```
⎕ES 'error-message'
⎕ES event-type
'error-message' ⎕ES event-type
```

In Dyalog APL, this is done with the Signal Event system function (⎕SIGNAL). It can be used in several ways:

```
⎕SIGNAL event-number
'error-message' ⎕SIGNAL error-number
⎕SIGNAL name-value pairs
```

In the third situation, the first element of the right argument is a 2-column matrix or a vector of 2-element vectors of name-values pairs, which are taken as a set of values to be used to override the default values in a new instance of ⎕DMX. For example:

```
      ⎕SIGNAL ⊂('EN' 200)('Message' 'Expired record')
ERROR 200: Expired record
      ⎕SIGNAL⊂('EN' 200)('Message' 'Expired record')
      ^
      ⎕dmx
 EM       ERROR 200
 Message  Expired record
```

Indexing

Dyalog APL supports both Choose and Reach indexing.

With Choose Indexing, the index specification is a non-simple array where each item identifies a single element of an array X by a set of indices with one element per axis of X in row-major order. For example:

```
      M←2 4 ⍴ 10 20 30 40 50 60 70 80
      M[(2 1)(1 2)]
50 20
```

With Reach Indexing, the index specification Y is a non-simple integer array, each of whose items reach down to a nested element of X. The items of an item of Y are simple vectors (or scalars) forming sets of indices that index arrays at successive levels of X starting at the top-most level. A set of indices has one element per axis at the respective level of nesting of X in row-major order. For example:

```
      G←('ABC' 1)('DEF' 2)('GHI' 3)('JKL' 4)
      G←2 3⍴G,('MNO' 5)('PQR' 6)
      G
ABC 1 DEF 2 GHI 3
JKL 4 MNO 5 PQR 6

      G[((1 2)1)((2 3)2)]
DEF 6
```

Internal Hash Tables for Search Functions

Dyalog APL supports retaining and reusing an internal hash table for certain search-related functions (e.g., Index of, Membership, Intersection, Union, Without, Matrix Iota (idiom) and Grade) by binding the function with its principal argument using the primitive operator Beside (⍤).

Retaining the hash table pays off only on a second or subsequent application of the derived function. This usually occurs in one of two ways: either the derived function is named for later (and repeated) use, or it is applied repeatedly as the operand of a primitive or defined operator. For example:

```
      words←'red' 'ylo' 'grn' 'brn' 'blu' 'pnk' 'blk'
      find←words⍤⍳                ⍝ monadic find function
      find'blk' 'blu' 'grn' 'ylo' ⍝
7 5 3 2
      find'grn' 'brn' 'ylo' 'red' ⍝ fast find
3 4 2 1
      ∈⍤⎕A¨'This' 'And' 'That'
1 0 0 0 1 0 0 1 0 0 0
```

Idioms

Dyalog APL supports Idioms, which are select commonly used expressions that are recognized by Dyalog APL and when found, are evaluated internally, providing a significant performance improvement.

APL Compiler

Dyalog APL supports an experimental compiler mechanism.

Enhanced Data Entry

Dyalog supports an enhanced data entry capability called Array notation which provides a simple way and easy way to enter most arrays. It allows for specification of vectors and arrays over multiple lines. In general:

- Parentheses surround vector definitions and namespace name-value pairs,
- Square brackets surround higher-rank arrays,
- Diamonds and line breaks separate array elements and name-value pairs, and
- A namespace literal can be written as zero or more name-valuepairs, spanned by parentheses.

Some examples

```
     ('One' ⋄ 'Two'
'Three')
```

One	Two	Three

```
     ['One' ⋄ 'Two'
'Three']
One
Two
Three
      n←(x:'hello')
      n.x
hello
```

Configuration Parameters

The behaviors of APL2 and Dyalog APL can be customized using different set of configuration parameters, some of which are environment dependent. These are not discussed here.

Calling APL

Each dialect supports different methods of calling APL from an external language.

Session Manager

The session manager (where available) is different between each dialect.

Dyalog APL allows for significant configuration of the session manager using APL including the appearance of the menus, tool bars, menu buttons and status bars, the addition new objects of a user's choice and attachment of a user's own APL functions and expressions to them. Functions and variables can be stored in a session namespace that is independent of the active workspace.

Sessions are stored in special files allowing for different sessions for different purposes.

Capturing System Commands

APL2 supports utilities that execute system commands via the stack and capture their results.

Dyalog APL does not.

Conformability

[This is AI material; someone who understands needs to validate this. It is noticeable that the dialects provide different extension rules in a number of functions and the behavior would be nice to document.]

APL2 defines general conformability with extension for many dyadic primitives:

- Conformability is axis-wise
- Axes are compared left to right
- Equal lengths match
- Length-1 axes are extended
- Scalars are a special case of length-1 arrays

This applies without any operator.

In Dyalog APL:

- Only scalars are extended
- A length-1 axis is not treated as scalar
- No axis-wise conformability exists
- Axes are not compared left to right for conformability
- Requires exact shape equality for non-scalars, with scalar extension as the sole exception
- Provides explicit control by using the Rank operator

APL-Coded User Screens

The user interfaces are different between the dialects. IBM APL2 is procedural while Dyalog APL is event driven.

Data Files

Data, or Component, files are accessed differently between the dialects.

Dyalog APL component files are modeled after the STSC / I.P. Sharp component file system which uses system commands created for this purpose (e.g., ⎕FAPPEND, ⎕FAVAIL, ⎕FCHK, ⎕FCOPY, ⎕FCREATE, ⎕FDROP, ⎕FERASE, ⎕FHIST, ⎕FHOLD, ⎕FLIB, ⎕FNAMES, ⎕FNUMS, ⎕FPROPS, ⎕FRDAC, ⎕FRDCI, ⎕FREAD, ⎕FRENAME, ⎕FREPLACE, ⎕FRESIZE, ⎕FSIZE, ⎕FSTAC, ⎕FSTIE, ⎕FTIE, ⎕FUNTIE).

APL2 supports a few file storage mechanisms, which vary based on platform:

AP 110	CMS File Processor (mainframe)
AP 111	QSAM File Processor (mainframe)
AP 121	APL2 Data File Processor (mainframe)

AP 123	VSAM File Processor (mainframe) (mainframe)
AP 210	BDAM File Processor (TSO)
AP 210	File Processor (workstation)
AP 211	APL2 Object File Processor
Processor 12	Files as Arrays

Auxiliary processors 110, 111, 123 and 210 for the mainframe allow for basic file manipulations for select file access types, as does AP 210 for the workstation.

AP 121 for the mainframe, and a subset of AP 210 functionality for the workstation, allow for storing APL objects in files, in a manner similar to storing objects in Component files in Dyalog APL. The commands used to do so vary between the APL2 platforms, and do not align with Dyalog APL.

AP 211 also stores APL objects but by name rather than by record number.

The auxiliary processors use shared variables (e.g., a control variable and a data variable) to access and manipulate files and the data within them. Encoded commands are passed in the control variable and associated data through a data variable (e.g., file names, data to store, read data).

Processor 12 uses name association (⎕NA) to associate a file with a name in an APL2 session. With Processor 12, APL files are viewed by APL2 as a vector of arbitrary arrays, with each item of the vector representing one object in the file. Access is done using normal APL functions and operations such as Each, Outer Product, Pick, Indexing, Indexed Assignment, Selective Assignment, Catenate, Shape, Compress/Replicate, First, Take and Drop. The file is not within the workspace, allowing for using large files that can be many times larger than the active workspace.

Processor 12 supports two file types. The first is an APL format file (compatible with AP 112 on the mainframe and AP 210 on the workstation). The second is a record-oriented file on the mainframe and a text format file on the workstation (compatible with AP 210 on the workstation). The record-oriented file on the mainframe has an extended description which is not explicitly identified as being compatible with other AP file types.

Unlike auxiliary processor shared variable associations, Processor 12 associations can be retained across)SAVE and)LOAD.

Leading Axis Theory

Leading Axis Theory is an approach to array language design and use that emphasizes working with arrays by manipulating their cells and mapping functions over leading axes implicitly using function rank or explicitly using the Rank operator. It is a feature of Dyalog APL but not of APL2.

It can be seen in major cell processing of an array, where a major cell is a sub-array on the leading dimension of an array X with shape 1↓⍴X.

For example, in Dyalog APL:

```
      X←3 4⍴⍳12
      X
1 2 3 4
5 6 7 8
```

```
9 10 11 12
      X⍳1 2 3 4
1
```

In APL2 this generates a RANK ERROR.

REFERENCES

1. -, https://forums.dyalog.com/viewtopic.php?f=13&t=1416
2. -, https://forums.dyalog.com/viewtopic.php?f=16&t=1914
3. -, APLWiki, https://aplwiki.com
4. -, APL2 Programming: System Services Reference, Version 2 Release 2, Second Edition, SH21-1054-01, IBM Corporation, March 1994.
5. -, APL2 Programming: Language Reference, Second Edition, SH21-1061-01, IBM Corporation, February 1994.
6. -, APL2 Language Summary, SX26-3851-02, IBM Corporation, 2016.
7. -, APL2 User's Guide, SC18-7021-23, IBM Corporation, 2017.
8. -, APL/PC Version 2.1, ZZ33-0523-0, IBM Corporation, 1986.
9. -, CASE STUDY: Automotive - Production Schedule, https://www.dyalog.com/case-studies/hercules.htm
10. -, Dyalog APL Language Reference Guide, Dyalog version 20.0, Dyalog Limited, 2025-10-30.
11. -, Dyalog APL Language - Migration Level, https://docs.dyalog.com/20.0/language-reference-guide/system-functions/ml/, Dyalog Limited.
12. -, Dyalog Binding Strengths Cheat Sheet, , Dyalog v20.0, https://docs.dyalog.com/20.0/files/CheatSheet_Binding_Strengths.pdf, Dyalog Limited, 2025-05-07.
13. -, Dyalog for Microsoft Windows UI Guide, Dyalog version 20.0, Dyalog Limited, 2025-10-30.
14. -, Dyalog for UNIX UI Guide, Dyalog version 20.0, Dyalog Limited, 2025-10-30.
15. -, Dyalog Programming Reference Guide, Dyalog version 20.0, Dyalog Limited, 2025-10-30.
16. - Parallel Language Features, Dyalog version 19.0, Dyalog Limited, 2023-07-14.
17. Gilgamesh Athoraya, Lift-Off from APL2 Mainframe to Dyalog in the Cloud: Migration of APL2 system from mainframe to Dyalog APL on Linux, https://www.dyalog.com/uploads/conference/dyalog22/presentations/U02_LiftOff_from_APL2_Mainframe_to_Dyalog_in_the_Cloud.pdf
18. Karl Holt, Morten Kromberg, Migrating APL+Win Applications, https://www.dyalog.com/uploads/conference/dyalog24/materials/D09_MigratingAPLWinApplications.pptx, Dyalog Limited, Glasgow, Scotland, September 2024.
19. Veli-Matti Jantunen, The Journey of an APL2 Bigot to Dyalog World, https://www.dyalog.com/uploads/conference/dyalog16/presentations/U04_Journey_of_APL2_Bigot_to_Dyalog.pdf, Glasgow, Scotland, 13 October 2016.
20. Morten Kromberg, Migrating APL+Win Applications, https://www.dyalog.com/uploads/files/presentations/APLGermany_autumn2024_MigratingAPLWinApplications.pptx. Dyalog Limited, Solingen, November 8, 2024.
21. Madeline Vergani RubenVerg, Omnibar: the ultimate language toolbar, https://omnibar.rubenverg.com/?ig=true&in=true&id=false&hid=false
22. Peter Simonsson, Jonas Stensiö & Lars Wentzel, Migrating from APL2 mainframe to Dyalog APL Windows, https://www.dyalog.com/uploads/conference/dyalog12/presentations/U25_APL2ToDyalog/Migrating%20APL2%20Mainframe%20to%20Dyalog%20APL(11).pptx,Elsinore, Denmark, from October 2012.
23. Rex Swain, Migration From APL2 to APL/W, Vector, the Journal of the British APL Association, Volume 12, No.4, https://www.rexswain.com/vector.pdf, British APL Association, April 1996, Pages 64 - 72.
24. Rexford H. Swain, Namespaces: APL/W vs. APL2, APL '95: Proceedings of the international conference on Applied programming languages, ACM SIGAPL APL Quote Quad: Volume 25 Issue 4, Pages 193–205, https://doi.org/10.1145/206913.207005, https://www.rexswain.com/apl95.pdf, Association for Computing Machinery, New York, NY, USA, June 1995.
25. Nancy Wheeler, Comparative file I/O methods in APL2, Proceedings of the International Conference on APL-Berlin-2000 Conference, ACM SIGAPL APL Quote Quad, Volume 30, Issue 4, Pages 241 - 246, https://doi.org/10.1145/570440.570506, Association for Computing Machinery, New York, NY, USA, 01 June 2000.